5-3-2021

Storing Intermediate Results in Space and Time: SQL Graphs and Block Referencing

Basem Ibrahim Elazzabi
Portland State University

Recommended Citation
Elazzabi, Basem Ibrahim, "Storing Intermediate Results in Space and Time: SQL Graphs and Block Referencing" (2021). *Dissertations and Theses.* Paper 5693.
https://doi.org/10.15760/etd.7566

Storing Intermediate Results in Space and Time:

SQL Graphs and Block Referencing

by

Basem Ibrahim Elazzabi

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
David Maier, Chair
Kristin Tufte, Co-Chair
Arash Termehchy
Antonie Jetter

Portland State University
2021

ABSTRACT

With the advancement of data-collection technology and with more data being available for data analysts for data-intensive decision making, many data analysts use client-based data-analysis environments to analyze that data. Client-based environments where only a personal computer or a laptop is used to perform data analysis tasks are common. In such client-based environments, multiple tools and systems are typically needed to accomplish data-analysis tasks. Stand-alone systems such as spreadsheets, R, Matlab, and Tableau are usually easy to use, and they are designed for the typical, non-technical data analyst. However, these systems are limited in their data-analysis capabilities. More complex data analysis systems provide more powerful capabilities, such as database management systems (DBMSs). However, these systems are complex to use for the typical data analyst and they specialize in handling a specific category of tasks. For example, DBMSs specialize in data manipulation and storage but they do not handle data visualization. As a consequence, the data analyst is usually forced to use multiple tools and systems to be able to accomplish a single data-analysis task.

The more complex and demanding the data-analysis task is, the more tools and systems are typically needed to complete the task. One monolithic data-analysis system cannot satisfy all data-analysis needs. Embracing diversity, where each tool and system specializes in a specific area, allows us to satisfy more needs than a monolithic system could. For example, some tools can handle data manipulation, while others handle different types of visualizations. However, these tools typically do not interoperate, requiring the user to move data back and forth between them. The result is a significant amount of time wasted on extracting, converting, reformatting, and

moving data. It would help to have a common client-side data platform that the data-analysis tools can all use to share their results, final and intermediate. Sharing intermediate results is especially important to allow the individual data-analysis steps to be inspected by a variety of tools. Moreover, sharing intermediate results can eliminate wasted computations by building on top of previous results instead of recomputing them, which can speed up the analysis process.

In this research we explore a new data paradigm and data model that allows us to build a shared data-manipulation system for a client-based data-analysis environment. In this shared system, we factor out the data manipulation process from data-analysis systems and tools (the front-end applications) into the shared system, leaving the front-end systems and tools to handle the unique tasks for which they are designed (e.g., visualizations). The shared system allows front-end applications to keep all or most of the intermediate results of their data-manipulation processes in main memory. The intermediate results can then be accessed and inspected by other front-end applications. This new data paradigm eliminates data movement between systems and significantly reduces unnecessary computations and repeated data-processing tasks, allowing the user to focus on the data-analysis task at hand. However there are significant challenges to implementing such a shared system.

Keeping all or most intermediate results in main memory is extremely expensive in terms of space. We present two novel concepts that we call *SQL Graphs* and *block referencing* that allow us to take advantage of two dimensions, space (main memory) and time (CPU), to store intermediate results efficiently. SQL Graphs are the data structure that we use to organize intermediate results, while block referencing is the mechanism that we use to store the data of these results. SQL Graphs and block referencing significantly reduce the space cost that is needed to store intermediate results and make our new data paradigm possible to operate on a client-based environment with limited capabilities (e.g., 8GB of RAM).

The main contributions of this research are as follows. We first describe and

explore the problem in question that data analysts face. We then introduce a new data paradigm to solve this problem. Then we explore the challenges that arise from implementing the new data paradigm. We then talk about the two new concepts, *SQL Graphs* and *block referencing* to solve the space-cost problem. Then we introduce another new structure that we call a *dereferencing layout index* (DLI) to solve the time-cost problem. We run experiments on these new techniques and concepts using a prototype of a system that we implemented called the *jSQL environment* (jSQL$_e$). We show our testing results and how effective the system is. We finally discuss some future work that can arise from this research and conclude this dissertation.

DEDICATION

To my parents whom without, I would not be where I am and this work would not have been possible. Thank you for all the hard work and the sacrifices that you have made.

"Be the change you wish to see in the world."

— *Mahatma Gandhi*

ACKNOWLEDGMENTS

First and for most, all thanks to God for his guidance and for allowing me to be who I am and where I am in life, and for allowing and enabling me to do the things that I do. Second, I would like to express my deepest appreciation and thanks to both of my advisors Professor David Maier and Professor Kristin Tufte. They were always a beacon of light for me throughout this difficult journey. They made it a lot easier and more fun and enjoyable than it would have been otherwise. The sheer amount of knowledge that I have acquired from both of them is priceless. I especially appreciate their patience with me and their generous support and commitment. So thank you both very much, because this work would not have been possible or have seen the light without you.

I also would like to thank my other committee members Professor Arash Termehchy and Professor Antonie Jetter for their interest in my work and for agreeing to be on the committee. Their encouragements and comments were valuable and were very much appreciated.

This work was supported by funding from TransPort (the Portland, Oregon regional coordinating committee for intelligent transportation systems), the Southwest Washington Regional Transportation Council (RTC), the Transportation Research and Education Center (TREC) at Portland State University, and the Intel Science and Technology Center for Big Data (ISTC-BD).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

There is a considerable amount of data analysis that takes place in client-based environments. A client-based environment is one that does not rely on a server computer. Many database systems are on server environments, usually with high-end computers, to be able to handle the heavy processing that these systems perform. In a server environment, a data analyst can then use a client application or tool running on a laptop or a desktop computer to send queries to the database system and get back the results.

There are many reasons why a data analyst would choose a client-based environment to analyze the data over a server-based environment. Not all data analysts have access to database systems where the data that they need resides. For those data analysts, the only means to acquire the data is through a medium such as exporting the data from the database, accessing the data through an application programming interface (API), or using a third-party application. In other cases, the tools that the data analyst uses to analyze the data, such as visualization tools, cannot use the database system directly and require the data to be fed to the tool in a specific format. Sometimes a server-based environment is too restrictive or slow for data analysis tasks, and analysts want to avoid the hassle of using a server environment. Other times data analysts simply want to work on the data offline. Finally, there are many data sets that are available on the Internet or other media that do not live in database systems, such as CSV or XML files. Although many data-set formats can be imported into a database system on a server environment, many data analysts choose to use less sophisticated and easier-to-use tools to perform the analysis on their personal computers.

1.1 CURRENT DATA-ANALYSIS SOLUTIONS

To analyze the data in a client-based environment, the analyst might use systems such as a spreadsheet, R [32], Matlab [34], or SAS [33]. However, such systems are restrictive and have limited capabilities. For example, spreadsheets have limits on the number of rows a data set can have. The R system is mainly designed for statistical analysis and Matlab is mainly designed for processing matrix and array mathematics. Both systems lack many data manipulation capabilities that can be found in database management systems. Although R and Matlab provide visualizations, the visualizations are not interactive and are basic compared to dedicated visualization tools. Systems such as Tableau [61] and Voyager [69] provide better, easy-to-use data-visualization capabilities, but provide few capabilities to the user to manipulate data. Tools such as D3 [11] and Vega [41] provide powerful and flexible data visualizations, but are less easy to use than the stand-alone ones and require the data to be prepared in a specific format.

Database management systems (DBMSs) provide powerful data-manipulation capabilities. However, visualizing the data requires third-party applications or tools such as Tableau [61], Zeppelin [6], or Jupyter [37] to connect to the DBMS. Other tools such as Vega [41] and D3 [11] need a more complicated process on the user's part to import the data into the tool to visualize it. For example, Vega requires the data to be in a JSON [19] format and D3 requires the data to be embedded as part of an HTML document or be provided using JavaScript code. Although DBMSs provide a shared system where, for example, multiple visualization tools can connect and visualize the same data, the data sharing happens at a low level. The same query might be executed multiple times by the DBMS, resulting in long execution times that do not satisfy application needs, such as interactive speed. Many tools try to compensate for the long execution time by pulling data at a low level (detailed data) and finishing the rest of the data manipulation process locally. As a consequence, the

intermediate results that one tool generates locally are invisible to others; inspecting these results can be difficult to impossible, depending on the tool.

Data-manipulation systems such as Spark [71], although less easy to use, provide a higher-level data-sharing environment by allowing the user to persist intermediate results in memory and share them across multiple applications. The persistence of intermediate results eliminates redundant computations across applications and allows data-analysis tools to collaborate and extract different insights from the data. However, keeping intermediate results in memory is expensive in terms of space, especially on a client-based environment where the typical RAM capacity nowadays is 8GB. The OS typically takes about 2GB of RAM, leaving less than 6GB to use for data analysis. Assuming that no other application is running on the machine, if we start with a data set with in-memory footprint of 2GB, performing two foreign-key joins is probably enough to consume the entire available memory for those intermediate results. Although the user can choose which results to persist and which ones to recompute, the user has to be strategic and be well aware of the space cost of each intermediate result, a task that is not suitable for a typical client-based data analyst.

1.2 THE IDEAL DATA-ANALYSIS SOLUTION

We realize that one-solution-fits-all does not exist, and possibly never will. Each data set and each decision have special data-analysis needs. Each data-analysis tool or system has advantages and disadvantages and has certain capabilities but lacks others. We believe that the closest we can get to fulfilling all variations of data-analysis needs is to have a data-analysis environment where multiple data-analysis tools and systems collaborate on analyzing the same data. The data analyst can then take advantage of a multitude of data-analysis capabilities that multiple tools and systems provide. However, such collaboration using existing data-analysis tools and systems is difficult and, sometime, impossible, especially for a typical data analyst. In many cases the analyst has to manually move the data back and forth between the

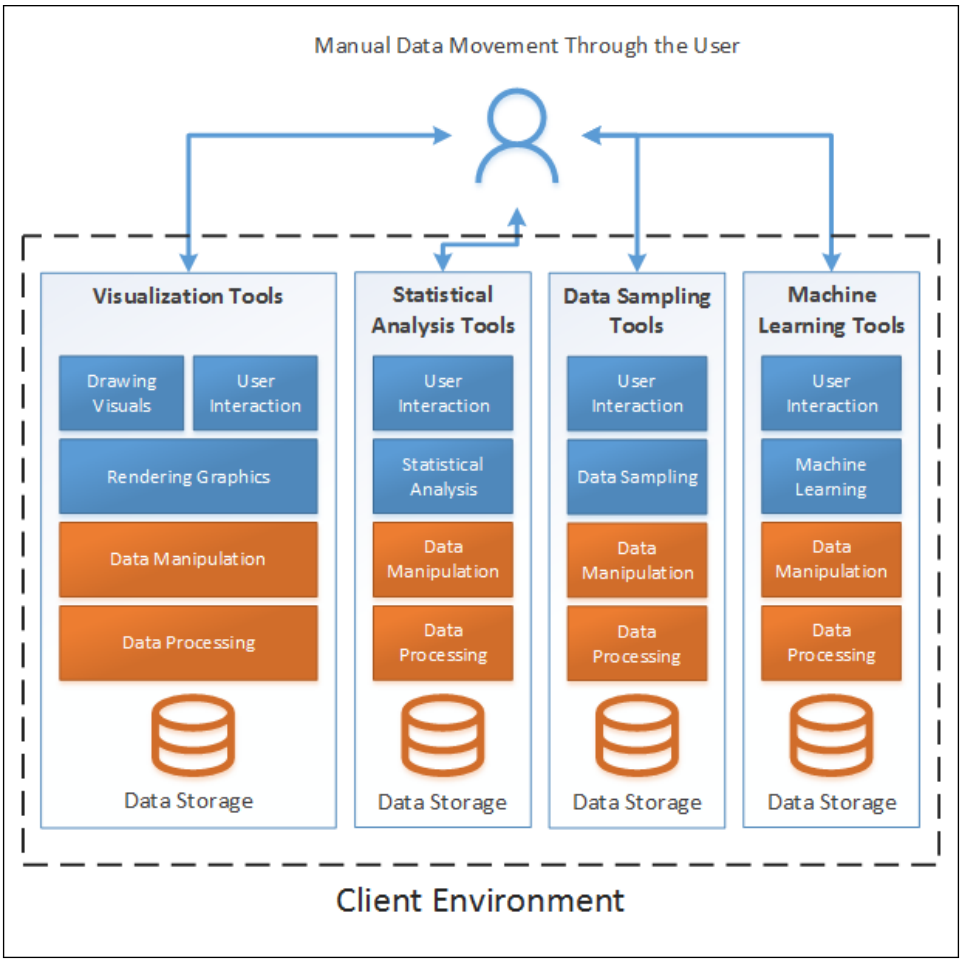


Figure 1-1: A data-analysis environment where each data-analysis tool manages its own data storage and performs data manipulation and processing locally. The user is forced to move data between tools manually by exporting the data from one tool and importing it into another.

tools (exporting the data from one tool and importing it into another), as illustrated in Figure 1-1. Sometimes the data that is exported from one tool is not compatible with the other, leaving the analyst to deal with data-format conversions. Moving data between various tools takes tremendous amount of time, effort, resources, and, in many cases, technical skills.

To enable data-analysis collaboration and, at the same time, eliminate data movement between data-analysis tools, we propose a shared, client-based data-analysis system where we factor out the data-manipulation process from these tools to allow

them to focus on the data-analysis parts that are unique to the tools themselves, as illustrated in Figure 1-2. Although DBMSs are shared data-manipulation systems, as we mentioned earlier, most of them provide sharing at a low level. Although various tools share the same base data, as illustrated in Figure 1-3, they do not share intermediate or final results. Such low-level sharing forces many tools to do part of the data-manipulation process in a DBMS and perform the rest locally. As a consequence of low-level sharing, the user is forced to manually move data (intermediate and final results that are processed locally) between various data-analysis tools. Moreover, this low-level sharing can result in repetitive computations across the tools and within the tools themselves.

To satisfy a wide range of data-analysis tools, the shared data-manipulation system must deliver data manipulation capabilities and data accessibility with interactive speed. Otherwise, many tools (e.g., visualization tools) will be forced to perform parts of the data-manipulation process locally to boost performance. Interactive speed is when a given tool interacts with the user within a tolerable time frame. For visualizations, interactive speed is usually defined to be around 500ms [43]. Ultimately, the exact value for interactive speed depends on the tool that is being used and the task at hand. For example, if we have two charts where dragging something on one chart changes the plots on the other, we need a time frame of 500ms or less. If we have a tool where we can click on a data set to display plots, a few seconds would be fine.

In addition to interactive speed, the shared data-manipulation system must also keep all or most intermediate results so that results (final or intermediate) generated by one tool can be accessed and inspected by another. For example, we might want to run statistical analysis on R [32] and then use Tableau [61] to plot the data (of final or intermediate results) to take advantage of interactive visualizations. We might also want the data that is being used in the plots to be displayed on some mapping platform such as Google Maps [26]. The analyst might also want to inspect the intermediate steps that led to the visualizations that are being generated by one tool and run some

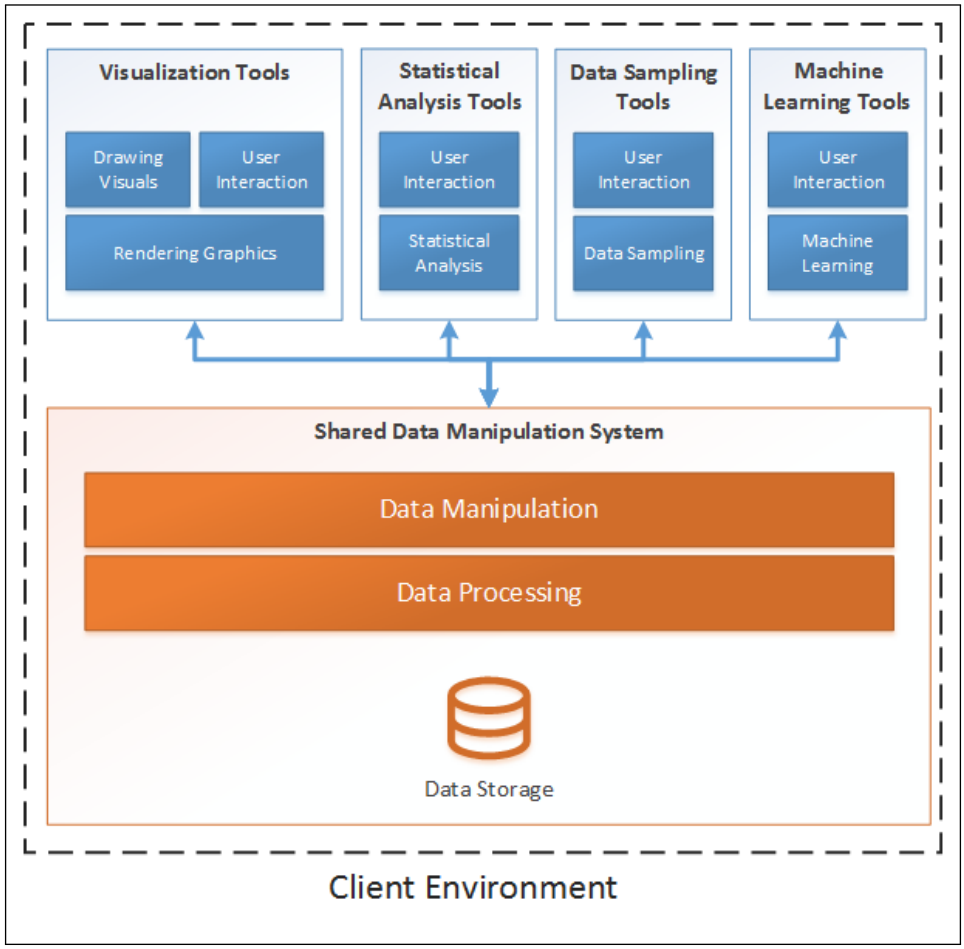


Figure 1-2: Our proposed data-analysis environment where the data storage, data manipulation, and data processing is factored out to a shared data-manipulation system. The user no longer needs to move data between tools since all the tools have access to the same data and the intermediate results generated by any of the tools.

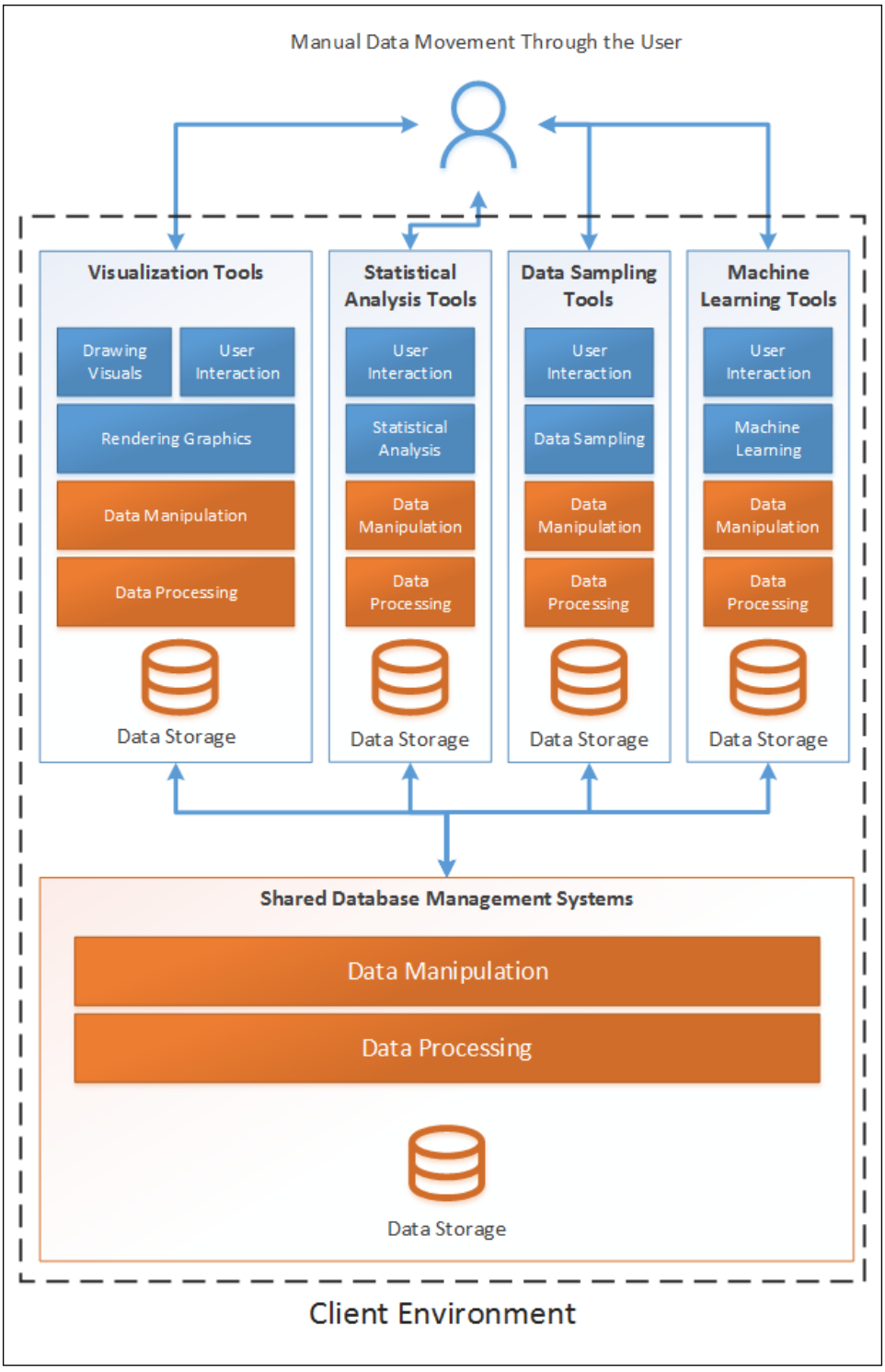


Figure 1-3: A data-analysis environment where each data-analysis tool delegates some of the data storage, data manipulation, and data processing to a database management system, while each tool still manages some of that responsibility locally. The user is still forced to move some data between tools manually since some of the data-manipulation process is done locally.

statistical analysis on R or explore other data-analysis paths starting from a given intermediate result. Although results can be recomputed when needed, recomputing results can be expensive, which prevents us from achieving interactive speed for many applications. Moreover, recomputation can consume user time, especially when two different tools need the same results (e.g., one tool displays the results on a map and the other on a chart).

1.3 CHALLENGES

As we discussed in the previous section, a shared data-manipulation system must provide interactive speed and must keep and store intermediate results. The problem is, however, keeping intermediate results can be expensive in terms of space (memory), especially for long data-analysis sessions. Although most disks nowadays are capable of storing such data, using disks can also prevent us from achieving interactive speed. Accessing data on disk is extremely slow compared to accessing the data in main memory (RAM). If we use main memory to store data as well as intermediate results, we can achieve interactive speed as a result of eliminating disk-access overhead. However, on a client-based environment, the size of main memory is far less than the size of disks (e.g., 8GB for RAM vs. 500GB for disk). The limited main-memory space that is shared across multiple applications provides little room for storing the data and performing data processing, let alone storing intermediate results, which are by far the most expensive in terms of space cost. Flash memory offers middle-ground performance[1] between disk and RAM. However, Flash memory is still not readily available in typical client-based environments, and it is not clear whether it would be capable of providing interactive speed.

During the data-analysis process, there is the space cost of storing the initial data sets with which we start the analysis. In addition, we now have the space cost of

[1]Flash memory is faster than disk but still significantly slower than RAM. In terms of space, Flash memory can be significantly larger than RAM but much smaller than disk, at least with respect to their cost.

storing intermediate results that are the product of the data-analysis process. For data analysis in a client-based environment, the initial data sets are typically small enough (within a few gigabytes) to fit in main memory. The size of intermediate results, however, can grow quickly depending on the operations themselves and the number of operations that are being used during the data-analysis process, making it impractical to store intermediate results as is in main memory. Compressing the data can achieve, in practical use cases, at most a compression ratio of 2× [47]. Some compression techniques [8, 9] can achieve 3–4× compression ratio for some use cases. However, for a relatively large (with respect to the size of the RAM) data set, a 2–4× compression ratio is not enough to store intermediate results for typical data-analysis use cases. Furthermore, we need to decompress the data before we are able to process it, which is an overhead that can prevent us from achieving interactive speed.

1.4 THE PROPOSED SOLUTION AND CONTRIBUTIONS

In this research, we propose a new data paradigm that allows us to build a shared data-manipulation system, as shown in Figure 1-2. Within this new paradigm, we explore a novel technique that we call **block referencing** that allows us to keep most or all intermediate results in main memory in addition to the initial data sets, while maintaining interactive speed. The general idea is to utilize two dimensions, space and time, to store data instead of the traditional one-dimensional, space-only approach of storing data. Throughout this research, we introduce and explore new concepts and techniques that allow us to efficiently keep data in main memory on a typical PC or laptop with 8GB of RAM and be able to keep tens and even hundreds of data-operator results in main memory. Specifically, our contributions are as follows.

1. We introduce a new data paradigm for a shared data-manipulation system that is able to keep all or most intermediate results in main memory and is able to provide interactive-speed data access to front-end data-analysis tools and systems.

2. We introduce the concept of **SQL Graphs**, a novel approach that allows us to organize intermediate results in a shared data-manipulation system.

3. We describe the mechanism behind storing data in two dimensions, space and time, to dramatically reduce the space cost of storing intermediate results.

4. We introduce the concept of **block referencing**, a novel data-storage approach that allows us to store intermediate results in the time dimension.

5. We introduce **data blocks**, a concept that allows us to find data-sharing opportunities to save space. Data blocks are also what we use to store data in the space dimension.

6. We explore the data-movement behavior of six common data operators: `select`, `project`, `join`, `union`, `group`, and `aggregate`. We identify three types of data-movement behaviors that allow us to find data sharing opportunities to save space cost.

7. We describe a general framework that allows us to store intermediate results at a low space cost (storing data in the space dimension) in exchange for a small CPU cost when we access the data (storing data in the time dimension). We also describe the algorithms that the six common operators use to achieve such space savings.

8. We introduce the concept of a **dereferencing layout index** (DLI), a novel approach that allows us to perform bulk dereferencing that achieves data-access time with interactive speed.

9. We describe a prototype that we built, which we call the **jSQL environment** (jSQL$_e$), for a shared data-manipulation system that meets and satisfies all the criteria we describe in this research.

10. We evaluate and explore the performance of our prototype and the techniques that are described in this research. Generally, we try to answer the following questions:

    (a) Does block referencing provide sufficient space savings compared to materialization to store intermediate results in a client-based environment?

    (b) What is the data-access-time cost of using block references?

    (c) Do DLIs improve data-access time?

    (d) How does our prototype that uses SQL Graphs and block references compare to other known, well developed data-manipulation systems in terms of space and time performance?

The rest of this dissertation is organized as follows. In Chapter 2 we describe the new data paradigm that we are proposing for the shared data-manipulation system. We talk about *SQL Graphs* and the data model in general, and we discuss the challenges that arise as we try to implement the model. In Chapter 3 we introduce the concept of *block referencing* to address these challenges and introduce a general framework for building block references. In Chapter 4 we talk about the space optimizations that we use for each data operator using block references. However, these space optimizations come at a time (CPU) cost that can prevent us from achieving interactive speed. In Chapter 5 we talk about time optimizations that use eager dereferencing and a new data structure that we call *dereferencing layout indexes* (DLIs). In Chapter 6 we discuss two approaches, a naïve approach and a space-efficient approach, for storing working data sets in main memory. In Chapter 7 we talk about the prototype system that we built using the concepts that we introduced in this research and we discuss the experiments and results that we did. In Chapter 8 we talk briefly about how we can add more data operators to this shared data-manipulation system to fulfill the needs of more front-end applications. In Chapter 9 we explore related

work and we explain how our work is distinct. Finally, in Chapter 10 we talk about some future work and conclude this dissertation.

CHAPTER 2: DATA PARADIGM

In Chapter 1, we discussed what we believe to be the ideal solution for a client-based data-analysis environment. Within this ideal environment, we argued that a shared data-manipulation system is necessary to improve data-analysis productivity. We also argued that for such a shared environment to exist, we must keep the data and intermediate results in main memory, and as well as provide data accessibility with interactive speed. In this chapter we propose and discuss an internal structure of this shared data-manipulation system. We first talk about the data model that we are going to use for this shared system. Then we discuss the challenges that arise from implementing this model. In the next few chapters, we will discuss concepts and techniques to address these challenges.

2.1 THE DATA MODEL

The data model of our shared data-manipulation system consists of *working data sets*, *data layers*, *data operators*, and the *SQL Graph*, as illustrated in Figure 2-1.

2.1.1 Working Data Sets

Working data sets are data sets with which the data analysis starts. Those data sets are usually extracted from a large database or found in some file format such CSV, XML, or JSON. The data sets can be loaded into the shared data-manipulation system in many ways, such as using a database driver (e.g., JDBC or ODBC) or using a URL to fetch the data. Once the data is loaded into the shared data-manipulation system, the data lives in its entirety in main memory, which means the initial data

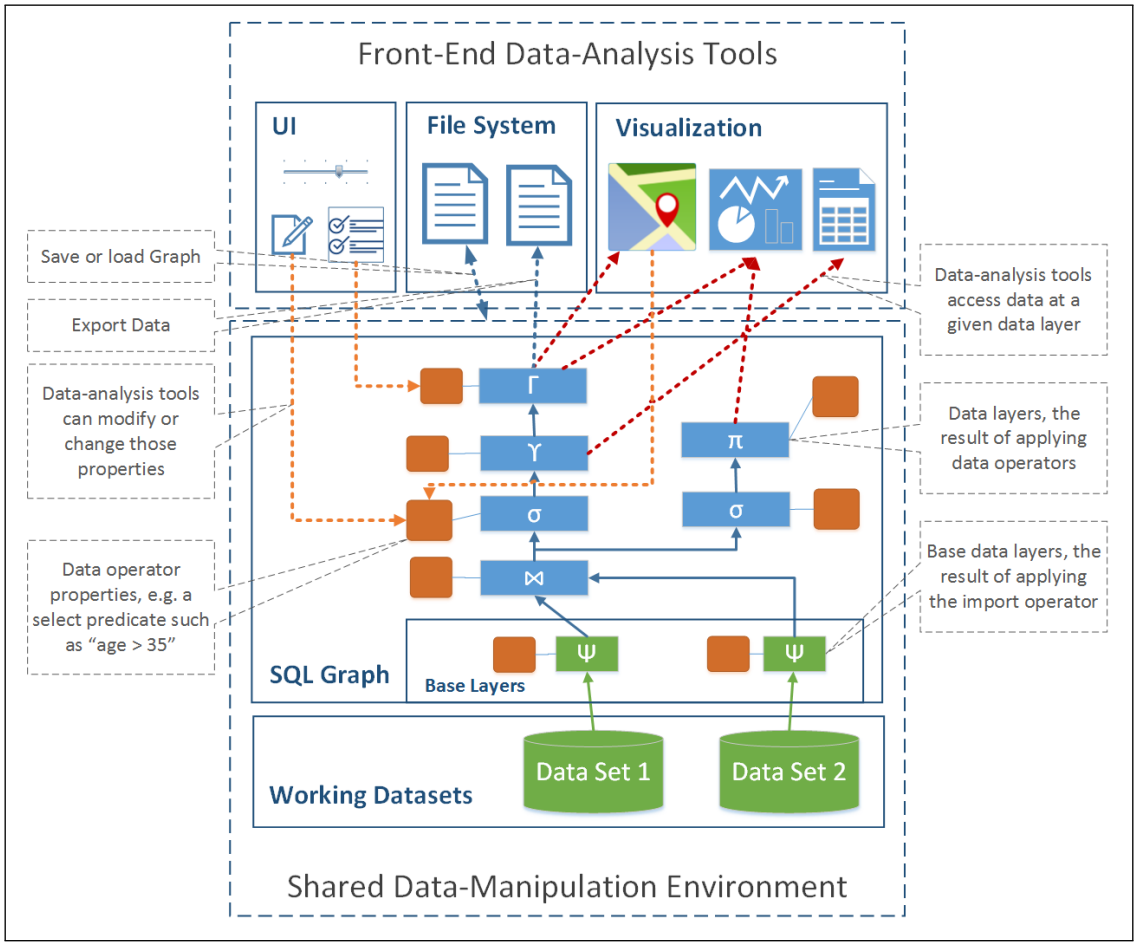


Figure 2-1: An illustration of the components of our data model inside the proposed shared data-manipulation system. The illustration also also shows how front-end data-analysis tools can connect to and interact with the system and share data.

set size must fit there. In this research, we do not focus much on the initial size of the data sets or how to reduce the space cost of keeping them in main memory. We assume that the initial size of the data sets fits in main memory and leaves some space for the rest of the data-analysis needs. We also assume that the data is read-only and no updates are needed during the data-analysis process.

2.1.2 Data Layers

Data layers are akin to relations in relational algebra and they are the result of the data operators in our model. In other words, data layers are how we capture intermediate results in our shared system. Data layers live in main memory and, unlike relations, are read-only and maintain the identity of the type of the operator that creates them. We refer to that identity as the *data layer's type*. For example, if a data layer is the result of a `select` operator, the data layer is called a `select` data layer and its type is `select`.

A data layer has two representations, a ***logical representation*** and a ***physical representation***. The *logical representation* is an interface that provides the conventional tabular view of the operator's result indexed by rows and columns. The *physical representation* is how the result of the operator is physically stored in main memory. There is a special type of data layer that we call a **base layer**. Although working data sets can be stored in any format, data operators in our model expect a tabular representation that can be accessed using a row i and a column j. A base layer acts as a wrapper for a working data set to provide a general interface to access the data using rows and columns, as expected by the data operators in our data model. In addition to the operator's identity, data layers also maintain references to the input layers, which ultimately creates the SQL Graph.

2.1.3 SQL Graph

The SQL Graph is the data structure that manages the intermediate and final results of data-manipulation expressions (a composition of data operations) in our model. The nodes of the graph are the data layers themselves, and the edges are the references that each data layer has to its operator's input layer(s). The SQL Graph starts with base layers (wrapping working data sets in a general data-layer interface). Then the SQL Graph grows as we apply data operators to existing data layers.

2.1.4 Data Operators

The data operators that we focus on in research are: `import` (ψ), `select` (σ), `project` (π), (`inner`) `join` ($\bowtie$), `union` ($\cup$), `group` (γ), and `aggregate` (Γ). However, the concepts and the framework we present can be extended to other data operators. Chapter 8 briefly talks about other operators that we have explored, such as `distinct` and other types of join, and other operators we introduce via algebraic equivalences. The following gives a brief description of the operators that we will use throughout this research.

- The `import` operator imports a working data set into the SQL Graph by wrapping the data set in a base layer.
- The `select`, `project`, `join`, and `union` operators are similar in functionality (not necessarily implementation) to those of relational algebra. The difference is that the operators in our model work with data layers and bags instead of relations and sets.
- The `group` operator groups the data based on a list of grouping columns and produces a set of groups of rows based on the values of the grouping columns, as illustrated in Figure 2-2. The number of rows in the output layer $\mathtt{L}_{out}$ depends on the number of unique values in the grouping columns. The schema of $\mathtt{L}_{out}$

consists of the grouping columns from the input layer $\mathtt{L}_{in}$ in addition to a new column that we refer to as group column. The data type of the group column is `collection` (a set of data rows).

- The `aggregate` operator aggregates a collection of rows based on a list of aggregation functions (e.g., `avg`, `min`, and `max`), as illustrated in Figure 2-3. In addition to the aggregation-function list, the operator takes as an input the collection column, which is of type `collection` (e.g., the group column that is generated by a `group` operator). The output layer $\mathtt{L}_{out}$ is a copy of the input layer $\mathtt{L}_{in}$ plus a new column for each aggregation function in the aggregation-function list to store the aggregation results from each function. There is another version of the `aggregate` operator where the collection column is not provided (or `null`), as illustrated in Figure 2-4. In such a case, the collection of rows on which the aggregations are performed is the entire $\mathtt{L}_{in}$. The output layer $\mathtt{L}_{out}$ contains only one row and a column for each aggregation function. For example, the function `COUNT` will count the number of rows in $\mathtt{L}_{in}$ if the collection column is not provided. On the other hand, if the collection column is provided, the `COUNT` function will count the number of rows in each group.

2.2 GOALS AND CHALLENGES

The goals that we want to achieve with the data model that we described in Section 2.1 are:

1. Keeping intermediate results in main memory for the duration of a data-analysis session.

2. Providing accessibility and data availability for these intermediate results to front-end applications and supporting cross-application data sharing.

3. Providing such data availability and accessibility within interactive speed.

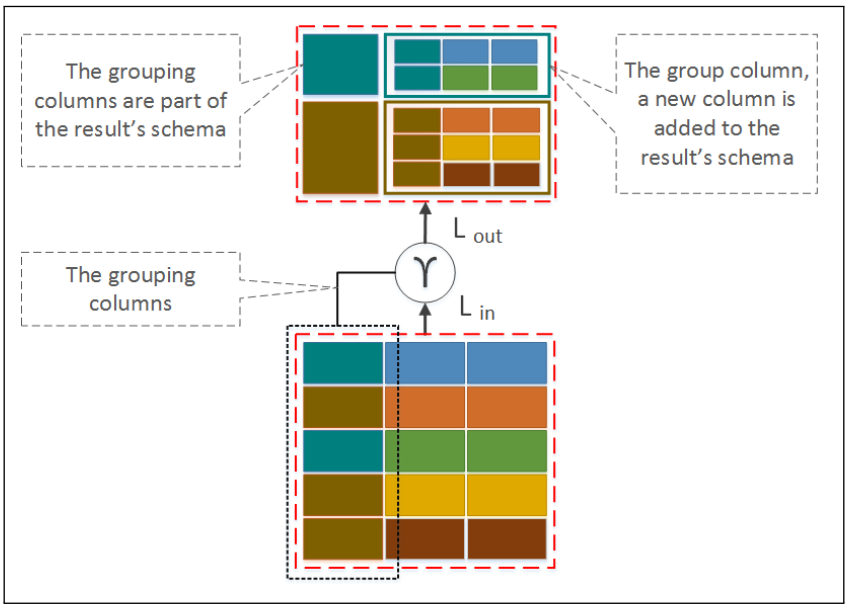


Figure 2-2: An illustration of how the `group` operator takes the data in an input data layer L_{in} and produces the output layer L_{out}. The schema of L_{out} consists of the grouping columns and a new column that is referred to as the group column.

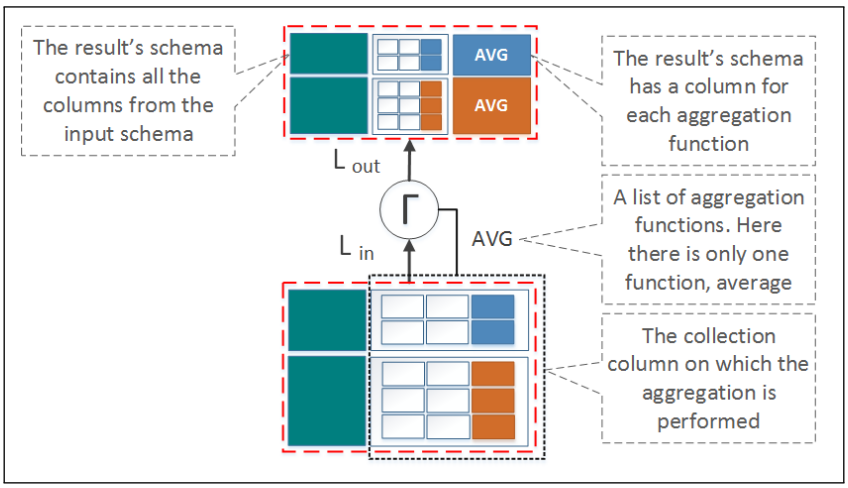


Figure 2-3: An illustration of how the `aggregate` operator takes the data in an input data layer L_{in} and produces the output layer L_{out} given a collection column. The schema of L_{out} consists of all the columns in L_{in} in addition to a column for each aggregation function.

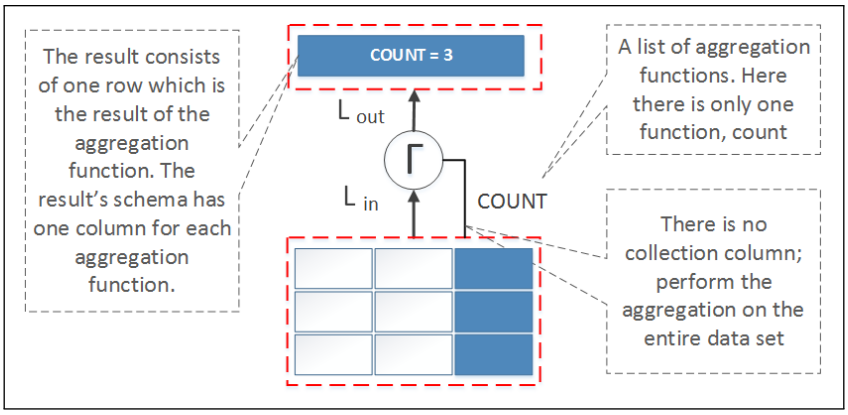


Figure 2-4: An illustration of how the `aggregate` operator takes the data in an input data layer L_{in} and produces the output layer L_{out} when a collection column is not given. The schema of L_{out} only has a column for each aggregation function.

We can achieve the first goal by using data layers; the challenge, however, is *the prohibitive space cost of keeping the intermediate results in memory*, especially when a data-analysis session (the SQL Graph) extends to tens or hundreds of data manipulations (data layers). We can achieve the second goal by accessing the logical representation of data layers. The logical representation provides a general interface that front-end applications can use to access the data in each layer regardless of how each data operator stores its results. To achieve the third goal, we need to overcome the challenge of providing logical-representation functions over physical representations with interactive speed.

To summarize the challenges:

- We need a significant space optimization to extend SQL Graphs to practical sizes without running out of memory for a typical PC (e.g., 8GB of RAM).
- At the same time, we need time optimizations to ensure data delivery to front-end applications with interactive speed.

Since we assume that the data are in main memory, all existing space-optimization techniques that we have explored, including data compression techniques, either have high CPU cost (usually above interactive speed), provide only marginal growth to SQL

Graphs (the extra saved space is enough for only a few more intermediate results), or both. In Chapter 3, we talk about a new approach that we call **block referencing** to overcome the first challenge and we show how we can use it to reduce the footprint of intermediate results dramatically. In Chapter 4, we talk about how to implement block referencing in the data operators we discussed in this chapter. In Chapter 5, we talk about a new structure that we call the **dereferencing layout index** (DLI) to overcome the second challenge and show how we can maintain interactive-speed data access within each data layer. In Chapter 6, we briefly discuss two approaches that we experimented with to store working data sets, a naïve, inefficient way and a more efficient way. Then in Chapter 7, we test the approaches and techniques that we discussed in the previous chapters. In Chapter 8, we briefly discuss how to add more data operators to our data model. In Chapter 9, we discuss some related work. Finally, we discuss some future work and conclude our research in Chapter 10.

CHAPTER 3: BLOCK REFERENCING

In Chapter 1, we discussed the importance of having a shared data-manipulation system in a client-based data-analysis environment. We also discussed the importance of keeping intermediate results in main memory as an essential requirement for such sharing to exist and for various front-end applications to be able to collaborate. In Chapter 2, we introduced a new data paradigm and a new data model that allows for such data sharing and collaboration to exist. However, one of the main challenges that we need to overcome is the prohibitive space cost of storing intermediate results, especially in main memory on a client-based environment. In this chapter, we introduce a new approach that we call **block referencing** that allows us to significantly reduce the space cost of intermediate results in the data model of Section 2.1.

Although we are not introducing new data operators in terms of data manipulation, we are introducing a new way for our operators to store their results (data layers) in main memory. The key idea that we focus on in this research is *finding data-sharing opportunities across data layers to reduce space and time costs*. We define a *data-sharing opportunity* as two or more data layers having an identical *data block* in their logical representations, where a *data block* (DBK) is a region of data cells; a *data cell* is a data unit holding a scalar value such as `75` or `"Bob"`. Sharing at the cell level is too small of a granularity because references are not appreciably smaller than the items they reference; thus we need something that represents larger chunks. There are four data block types (Figure 3-1) in which we are interested: a *data row* (DR), a *data column* (DC), a *range of data rows* (RDR), and a *range of data columns* (RDC).

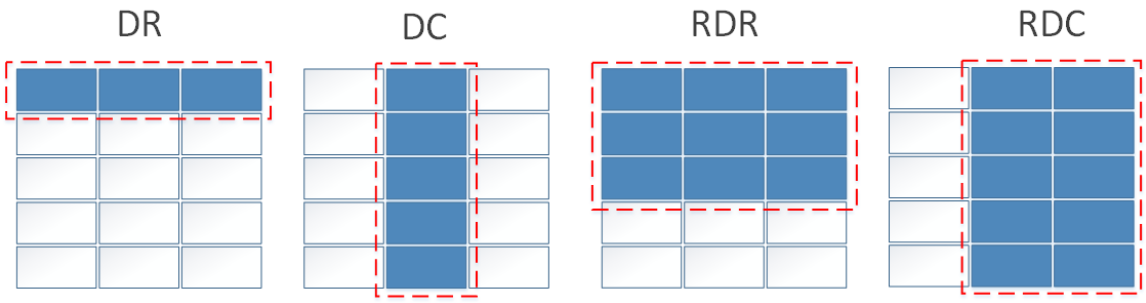


Figure 3-1: Data-block types (from left to right): data row, data column, range of data rows, and range of data columns.

The specific interest in these four data-block types is driven by how data behaves[1] when it moves through operators from the input data layer(s) to the output layer. In order to understand how data-sharing opportunities manifest across data layers, we first need to understand data-movement behavior across various data operators. For the rest of this chapter, we first discuss the classes of data-movement behaviors that we observed in the data operators that we use and discuss how data-sharing opportunities arise. Then we talk about the general idea behind block referencing and how we can use it to save space cost.

3.1 DATA-MOVEMENT BEHAVIOR

When data flows through operators (specifically the ones we use in this research), we observe three general classes of behaviors: *redundancy behavior* (RUB), *origin-generation behavior* (OGB), and *order-preserving behavior* (OPB). Note that these classes of behavior are not mutually exclusive. With each behavior class, we want to know how much we have to pay in terms of space (RAM) and time (CPU) to access the data at the output layer. Also, if we see one of these behavior classes, we want to know if we can pay a small amount of dereferencing time in exchange for large space savings for not materializing the results (storing data in time instead of space). The

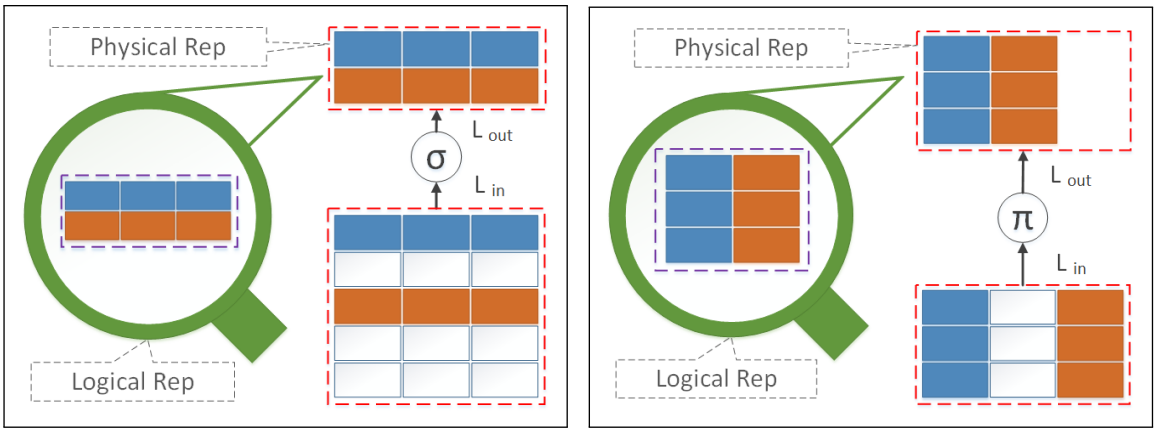
[1]For the data operators that we studied, the behavior that we see is that a group of data cells with shared properties move through an operator together. Because of such behavior, we can express those groups in terms of these shared properties.

following is a description of each of the behavior classes. We discuss each operator's behavior in detail in Chapter 4.

- Redundancy Behavior (RUB): This behavior arises when a data block from the input layer is replicated in the output layer. The replicated data blocks create a data-sharing opportunity between the input and the output layers. Instead of replicating the data blocks, we need to make the input and the output layers share those blocks, thus eliminating the extra space cost. We discuss the sharing mechanism in the next section. There are two types of redundancy behaviors that we see in some of the operators: *copy row* (CR) and *copy column* (CC). An example of a copy-row behavior can be seen in the `select` operator, as illustrated in Figure 3-2a, while Figure 3-2b shows an example of a copy-column behavior for the `project` operator.

- Origin-Generation Behavior (OGB): This behavior arises when an operator generates a new data block. We call such new data blocks *origin blocks*. When origin blocks are created, we have to pay full price either in terms of time or space. For example, we can pay the full price in time by running the query or computing the expression on the fly every time we want to access the data in that block. On the other hand, we can pay the full price in space by simply caching the contents of the block in the output layer. There are two types of origin-generation behaviors that we see in some of the operators: *generate row* (GR) and *generate column* (GC). An example of a generate-row behavior can be seen in the `aggregate` operator without a collection column, as illustrated in Figure 2-4, while Figure 2-3 shows an example of a generate-column behavior for the `aggregate` operator with a collection column (generating a new column for each aggregation function).

- Order-Preserving Behavior (OPB): This behavior arises when an operator's implementation preserves the order of the row or the column index of the moved



(a) Copy-row (CR) behavior in a `select` operator

(b) Copy-column (CC) behavior in a `project` operator

Figure 3-2: The redundancy behaviors in `select` and `project`.

data blocks with respect to the input layer. Formally speaking, an operator OP with a given implementation *imp* is *order-preserving* if the following is true:

> Given any two blocks b_1 and b_2 in the input layer L_{in}, if we can access the two blocks in L_{in} using the row or the column indexes i and j, respectively, where $i < j$, it must be the case that we can access the same two blocks (if they propagate) in L_{out}, where $\mathsf{L}_{out} = OP_{imp}(\mathsf{L}_{in})$, using some indexes m and n, respectively, where $m < n$.

There are two types of order-preserving behaviors that we see in some of the operators: *row order-preserving* (ROP) and *column order-preserving* (COP). Preserving order (row or column) enables more sharing opportunities as we will see in Chapter 5. Both row-order and column-order preserving behaviors can be seen in the `select` operator. Although the row-order preserving behavior in the `select` operator depends on the implementation of the `select` algorithm, there is nothing inherent about the operator's behavior that prevents us from reordering the rows to preserve the order, unlike the `join` operator.

Now that we understand how sharing opportunities manifest, next we explain

the data-block sharing mechanism (block referencing) that will enable us to keep intermediate results in main memory efficiently.

3.2 SHARING DATA BLOCKS USING BLOCK REFERENCING

When there are two identical data blocks, one is in the input layer L_{in} and the other is in the output layer L_{out}, we want both layers to share one physical block instead of having a separate block for each layer. The main principle on which the sharing mechanism relies is as follows: for a given origin data block d, instead of replicating d, we want to create a block reference p to d, as illustrated in Figure 3-3, such that:

$$SC(p) < SC(d), \qquad \text{(Condition (1))}$$

$$\text{and} \quad TC(p) < \mathit{Interactive\ Speed}, \qquad \text{(Condition (2))}$$

where SC is the space cost (memory) and TC is the time cost (CPU) of dereferencing p to retrieve data values from d. The value for interactive speed depends on the application. For example, visualization tools usually define interactive speed between 500ms and 1sec. When we present our experiments in Chapter 7, we will specify the value for interactive speed. For now, we define it as the value that a given application can tolerate as data access time.

The reason why the principle works within our data model is because data layers are read-only. That is, we know that origin data blocks will never change with respect to the replicated data blocks. In DBMSs, updating data is a necessary feature, and they must account for the replicated data blocks to be modified at any time.

A **block reference** is information that tells us how to find the data block d in memory. Block referencing allows us to store data in the time dimension, whereas data blocks allow us to store data in the space dimension. The idea for reducing space cost is to "store" data in time (pay CPU cycles to dereference p) instead of

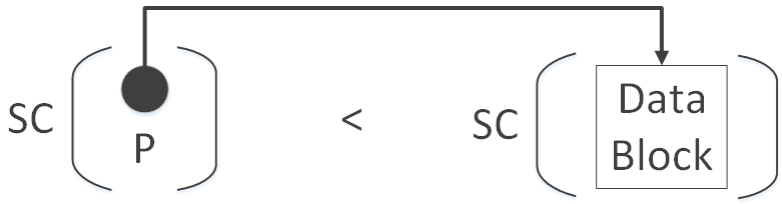


Figure 3-3: The goal that block references must accomplish is that the space cost (SC) of the pointer must be less than the cost of the data block it references.

storing data in space (by replicating the block), as long as the time cost is within interactive speed. From Condition (1), to achieve high space-cost savings, we need to maximize the *space-cost difference* (SCD); that is, we need $SC(d) - SC(p)$ to be as large as possible, while satisfying Condition (2). In Chapter 4 and 5, we discuss how to maximize this space-cost difference. Chapter 4 focuses on space optimizations that maximize Condition (1). Then Chapter 5 discusses time optimizations to satisfy Condition (2).

CHAPTER 4: SPACE OPTIMIZATIONS

Chapter 2 introduced a data model that enables a shared data-manipulation system in a client-based data-analysis environment. Chapter 3 introduced block referencing and discussed the general mechanism that allows us to build such a shared system. We specifically discussed that to use block referencing effectively, we need to satisfy two conditions: (1) the space cost of a block reference must be less than the space cost of the data block it references and (2) the time cost to dereference a block reference must be less than interactive speed. In this chapter, we focus on satisfying Condition (1), while Chapter 5 focuses on Condition (2).

Whenever we apply an operator, we store its result in a data layer. As we mentioned in Section 2.1.2, a data layer has two representations, a logical representation and a physical representation. The logical representation is what the user or the application sees, whereas the physical representation is how the data is physically stored. The idea is to use block references, as much as possible, as the physical representation to store results instead of the actual data, as long as we satisfy Condition (1). However if we use block referencing, the physical representation now is not structurally equivalent to the logical representation that the user or the application expects. So we need a mechanism that translates a physical representation to the expected logical representation.

Block referencing relies on two main functions: `build()`, which determines how to construct block references for a given layer to build its physical representation, and `getValue()`, which determines how to dereference these block references with respect to the logical representation, using the physical representation. The goal is to come up with an implementation for `build()` that satisfies and maximizes Condition (1)

and an implementation for `getValue()` that satisfies Condition (2). There is also the question of the time cost for `build()`. The answer is, the time cost for `build()` consists of two costs, 1) the time cost to access the data to be processed and 2) the time cost to run the operator's algorithm to process the data. Since `getValue()` is responsible for accessing data, we already covered cost 1). In this research, we do not talk about how to write algorithms to process data fast for various types of data operators (the database literature is full of such research). How fast the data can be processed (once we have access to the data) for a given operator depends, among others, on which algorithm you choose and how good the system's query optimizer is, neither of which is the focus of this research.

Although we can design a general implementation for each function for all operators, we realized that considering each operator individually allows us to substantially increase the space-cost difference between the block reference and the data block that it references. The data layer's type provides an implicit context to the meaning of a block reference in a layer of that type and a context to how it should be dereferenced to reach the actual data values. This implicit context provides more sharing opportunities among data layers of the same type and reduces the space cost that is needed for a block reference.

The function `build()` is a *function of the operator's class*, which takes the input layer(s) ($\mathtt{L}_{in}$), possibly with other operator-specific parameters, and returns an instance of a data layer of the same type as the operator. A data layer instance has three main attributes: the input layer(s) $\mathtt{L}_{in}$ (which we need to access blocks in those layers), the `schema` (a list of pairs of field name and data type), and the physical representation `data` (the contents of this attribute depend on the operator). The function `getValue()` is a *function of the data-layer instance*, which takes a data cell's row i and column j (w.r.t the logical representation of the layer) and returns its value as expected by the user or the application.

In Section 4.1, we talk about a space-efficient implementation of both functions

for each operator. To help understand the concepts, we will use a *simplified model* throughout this chapter. The simplified model assumes that only base layers can be inputs to operators. Note that base layers can only be created using the `import` operator to wrap a working data set into a data-layer interface. The reason why we only allow base layers to be inputs to operators is because the physical representation of a base layer is structurally equivalent to its logical representation. This equivalence simplifies the algorithms significantly and allows us to focus on explaining the concept of using block referencing. In Chapter 5, we show how we can extend the algorithms and the techniques that we use in this chapter to support the full model (a full SQL Graph) where input layers can be the result of any operator. In Section 4.2, we analyze the space cost of each operator's implementation and talk about how much space we save by using block referencing.

4.1 THE OPERATOR IMPLEMENTATIONS

4.1.1 `Import`

The `import` operator simply wraps a working data set in a data-layer interface; the output of `import` is a base layer. The idea of `import` is to provide an interface where the data in a working data set can be accessed using a row i and a column j. As long as we can build such an interface, the data in the working data set can be in any format, though different formats result in different access-time costs. Since building the interface is not the focus of this research, for simplicity, we are going to assume that the working data set is a two-dimensional array (`array`) where each column can have its own data type. The following are the `build()` and `getValue()` algorithms.

```
1 function build(array, schema)
2   return new ImportLayer{L_in: null, schema: schema,
3       data: array}
```

```
1 function getValue(i, j)
2   return this.data[i][j]
```

4.1.2 `Select`

The conventional behavior of a `select` operator is to replicate (CR, copy-row behavior) in the output layer ($\mathtt{L}_{out}$) the rows that satisfy the predicate from the input layer ($\mathtt{L}_{in}$). For our implementation of the `select` operator, we also assume that the operator keeps the rows in the order they appear in $\mathtt{L}_{in}$[1] (ROP, row-order preserving behavior) as shown in Figure 4-1a. Since the data-row (DR) blocks in $\mathtt{L}_{out}$ are already present in $\mathtt{L}_{in}$, to save space, we will use block references to point to the original DR blocks instead of replicating them. There are two things to notice:

1. The schema in $\mathtt{L}_{out}$ equals the schema in $\mathtt{L}_{in}$. Since `select` does not affect columns or their data types, the schema stays the same.

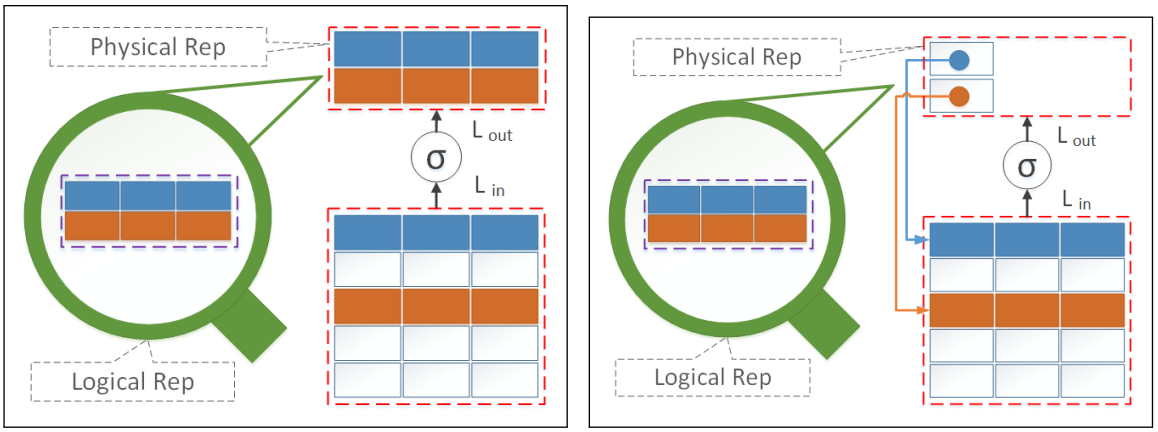
2. The replicated rows are not necessarily contiguous. For example, DR blocks 1 and 5 in $\mathtt{L}_{in}$ might become DR blocks 1 and 2 in $\mathtt{L}_{out}$ as a result of filtering out DR blocks 2 to 4 from $\mathtt{L}_{in}$. However, as we mentioned, the order stays the same. That is, if DR block i comes before DR block j in $\mathtt{L}_{in}$, it will still be the case in $\mathtt{L}_{out}$ because we can maintain the order[2].

Instead of storing the entire DR blocks in $\mathtt{L}_{out}$, the only information that we need to store to retrieve a given data value is the indexes of the DR blocks from $\mathtt{L}_{in}$ that satisfy the predicate. So the block references in a `select` data layer can be integers that represent indexes of DR blocks in $\mathtt{L}_{in}$, as shown in Figure 4-1b. The following are the `build()` and `getValue()` algorithms.

```
1 function build(L_in, predicate)
2   rowIndexes = []
```

[1]Preserving order is an important property that will become useful later in Chapter 5 when we perform time optimizations. Specifically it will determine whether the operator can have a DR implementation (see Section 5.2.1) or not.

[2]Even if blocks 1 and 5 in $\mathtt{L}_{in}$ are duplicates, in which case we cannot tell which is which in $\mathtt{L}_{out}$, the statement (the blocks are in the same order in $\mathtt{L}_{out}$ as they are in $\mathtt{L}_{in}$) is still valid. We can map block 1 in $\mathtt{L}_{in}$ to either block 1 or 2 in $\mathtt{L}_{out}$ and, similarly, map block 5 in $\mathtt{L}_{in}$ to either block 1 or 2 in $\mathtt{L}_{out}$. Since we want to preserve order, we can pick the mapping where block 1 in $\mathtt{L}_{in}$ maps to block 1 in $\mathtt{L}_{out}$ and block 5 in $\mathtt{L}_{in}$ maps to block 2 in $\mathtt{L}_{out}$.



(a) A `select` operator that uses replication (b) A `select` operator that uses references

Figure 4-1: A comparison between a `select` operator where data is replicated and a one where data is referenced.

```
for i in [0 ... L_in.size()-1]
  if predicate(L_in, i)
    rowIndexes.add(i)
return new SelectLayer{L_in: L_in,
    schema: L_in.schema, data: rowIndexes}
```

```
function getValue(i, j)
  i' = this.data[i]
  return this.L_in.getValue(i', j)
```

4.1.3 `Project`

Although `project` selects columns (CC, copy-column behavior) from the input layer $\mathtt{L}_{in}$ as well as generates columns (GC, generate-column behavior) using calculated columns (e.g., $\pi_{x+y}(\mathtt{L}_{in})$), we restrict the behavior of a `project` operator to only selecting columns from $\mathtt{L}_{in}$ for this chapter and the next. In Chapter 8, we briefly discuss how we were able to extend the `project` implementation to include calculated columns. The conventional behavior of a `project` operator is to replicate the selected columns from $\mathtt{L}_{in}$ in $\mathtt{L}_{out}$ as shown in Figure 4-2a. Since the data-column (DC) blocks

already present in L_{in}, to save space, we will use block references to point to the original DC blocks instead of replicating them. There are two things to notice:

1. The schema in L_{out} consists of only the selected columns.
2. The `project` might permute the columns in a different order, but the values into them are in their original row order.

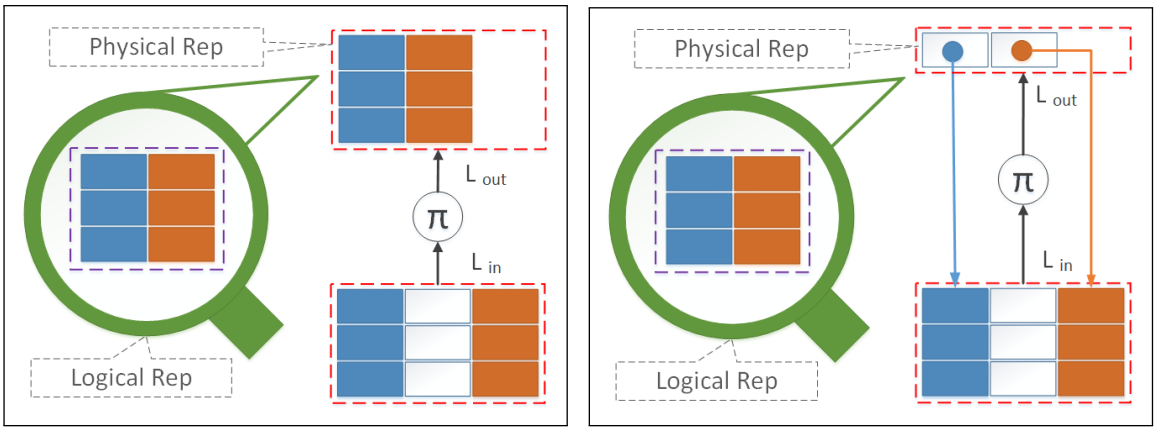
Instead of replicating the entire DC blocks in L_{out}, the only information that we need to store to retrieve a given data value is indexes of the DC blocks that are being projected from L_{in}. So the block references in a `project` data layer can be integers that represent indexes of DC blocks in L_{in}, as shown in Figure 4-2b. The following are the `build()` and `getValue()` algorithms, given a list of projected column indexes (`colIndexList`) from L_{in}:

```
function build(L_in, colIndexList)
  schema = []
  for j in colIndexList
    schema.add(L_in.schema[j])
  return new ProjectLayer{L_in: L_in, schema: schema,
    data: colIndexList}
```

```
function getValue(i, j)
  j' = this.data[j]
  return this.L_in.getValue(i, j')
```

4.1.4 `Union`

The conventional behavior of a `union` operator takes the contents of L_{in2}, appends it to the contents of L_{in1}, and replicates the result (CR, copy row, and CC, copy column, behaviors) in L_{out} as shown in Figure 4-3a. In addition, the operator maintains the order of the rows and the columns (ROP, row-order preserving, and COP, column-order preserving, behaviors). Since the entire contents of both input layers



(a) A `project` operator that uses replication

(b) A `project` operator that uses references

Figure 4-2: A comparison between a `project` operator where data is replicated and a one where data is referenced.

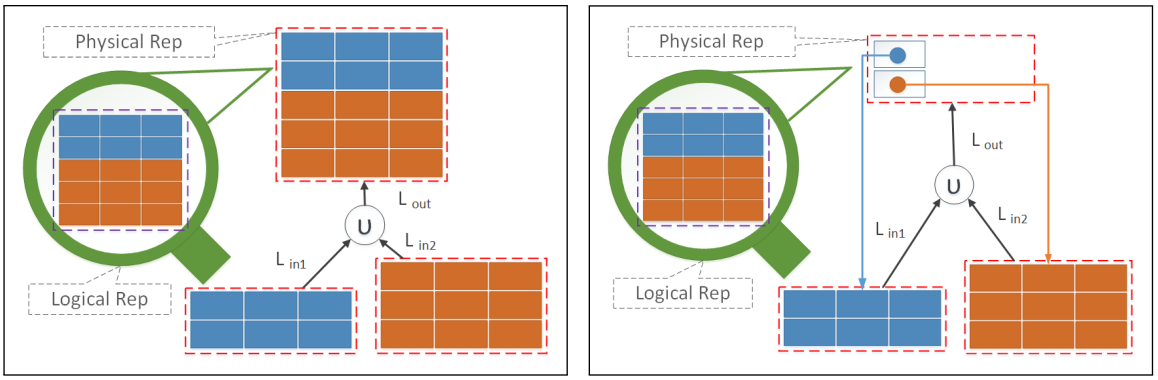
are replicated, we can consider each input layer as one data block (RDR block, a range of data rows, where the range is from 0 to $n-1$, and where n is the number of rows in the layer). Instead of replicating the RDR blocks in $\mathtt{L}_{out}$, to save space, we can use block references to point to the original ones in the input layer. There are two things to notice:

1. Following the SQL standard, the schema in $\mathtt{L}_{out}$ equals that of $\mathtt{L}_{in1}$. At the same time, $\mathtt{L}_{in2}$'s schema must be compatible with $\mathtt{L}_{in1}$'s schema in terms of the number, order, and data type of attributes or fields.

2. The rows and columns in the RDR blocks are in their original order.

Instead of storing the entire RDR blocks in $\mathtt{L}_{out}$, the only information that we need to store in $\mathtt{L}_{out}$ to access a given data value in the RDR blocks from the input layers is the start-row indexes (`startRowIndexes`) at which we need to use $\mathtt{L}_{in1}$ or $\mathtt{L}_{in2}$. So the block references in a `union` data layer are two integers, each of which represents a RDR block from one of the input layers, as shown in Figure 4-3b. The following are the `build()` and `getValue()` algorithms:

```
1 function build(L_in1, L_in2)
```



(a) A `union` operator that uses replication (b) A `union` operator that uses references

Figure 4-3: A comparison between a `union` operator where data is replicated and a one where data is referenced.

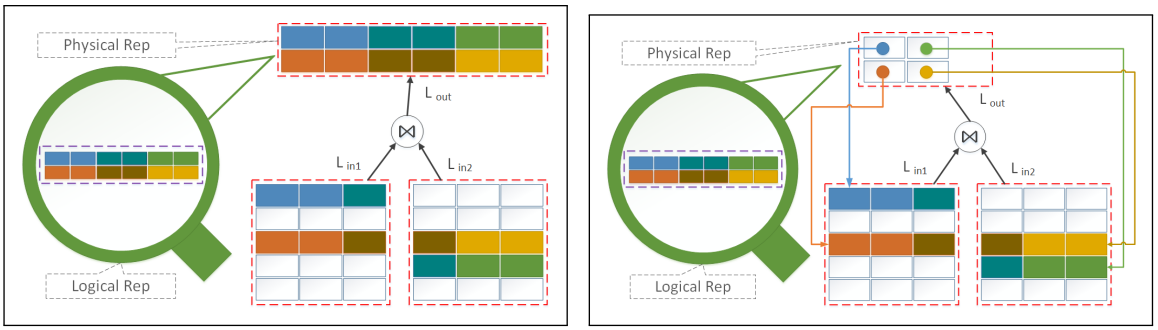
```
    startRowIndexes = [0, L_in1.size()]
    return new UnionLayer{L_in1: L_in1, L_in2: L_in2,
      schema: L_in1.schema, data: startRowIndexes}

function getValue(i, j)
    if i < this.data[1]
      return this.L_in1.getValue(i, j)
    return this.L_in2.getValue(i − this.data[1], j)
```

4.1.5 `Join`

The conventional `join` (inner join) operator matches rows in $\mathtt{L}_{in1}$ with rows in $\mathtt{L}_{in2}$ based on a predicate and replicates in $\mathtt{L}_{out}$ the pair of matching rows (CR, copy row, behavior) from both inputs, as shown in Figure 4-4a. Since the matching data-row (DR) blocks already present in the input layers, to save space, we will use block references to point to the original DR blocks instead of replicating them. That is, instead of having a pair of DR blocks in $\mathtt{L}_{out}$ for the matching rows, we will have a pair of block references, one for each DR block. There are two things to notice:

1. The schema in $\mathtt{L}_{out}$ is the concatenation of $\mathtt{L}_{in1}$ and $\mathtt{L}_{in2}$'s schemas.

2. The replicated rows are not necessarily in their original order.



(a) A `join` operator that uses replication (b) A `join` operator that uses references

Figure 4-4: A comparison between a `join` operator where data is replicated and a one where data is referenced.

Instead of storing the pair of DR blocks in L_{out}, the only information that we need to store to retrieve a given data value is a pair of indexes of the matching DR blocks. So the block references in a `join` data layer are pairs of integers, each pair represents indexes of two DR blocks, one from L_{in1} and another from L_{in2}, as shown in Figure 4-4b. The following are the `build()` and `getValue()` algorithms.

```
function build(L_in1, L_in2, predicate)
  indexPairs = findMatches(L_in1, L_in2, predicate)
  schema = L_in1.schema.clone().append(L_in2.schema)
  return new JoinLayer{L_in1: L_in1, L_in2: L_in2,
    schema: schema, data: indexPairs}
```

```
function getValue(i, j)
  pair = this.data[i]
  l2StartCol = this.L_in1.schema.size()
  if j < l2StartCol
    return this.L_in1.getValue(pair[0], j)
  return this.L_in2.getValue(pair[1], j − l2StartCol)
```

4.1.6 `Group` (γ)

As shown in Figure 4-6a, a `group` (γ) operator groups the rows in L_{in} based on a given list of grouping columns (`groupColList`). For more on how the `group` operator works,

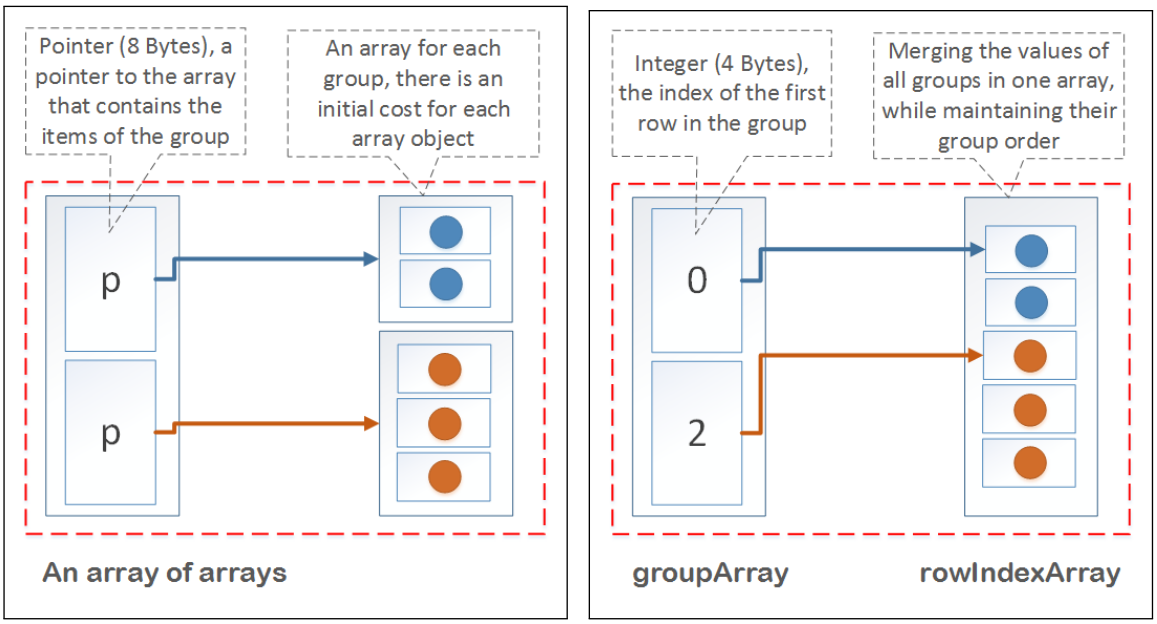
see Section 2.1.4. The rows in $\texttt{L}_{in}$ are then replicated (CR, copy row, behavior) and put in groups, which themselves are stored in a list in $\texttt{L}_{out}$. Notice that:

1. The schema in $\texttt{L}_{out}$ consists of the grouping columns in addition to the group-list column whose name is given by the user (`groupColName`). The data type of the group-list column is `collection<S>`, where `S` is some schema. That is, each value in the group-list column is a group of rows, all of which have the same schema `S` (or a schema that is compatible with `S`).

2. For any row in $\texttt{L}_{out}$, the values of the grouping columns can be obtained from any row in the group column.

The information that we need to store in $\texttt{L}_{out}$ to retrieve a given data value is the row indexes from $\texttt{L}_{in}$. However, we need a mechanism to figure out which indexes belong to which group.

We can store the indexes as an array of arrays, each of which is an array of row indexes in $\texttt{L}_{in}$ representing the rows in a given group, as illustrated in Figure 4-5a. However, this structure is not space efficient, because we need to create an `array` object for each group (the initial array size varies from one language to another) and we need an additional 8 bytes for an `array` pointer for each group; the `array` pointer is then stored in the group array. A more efficient structure is to use one array (`rowIndexArray`) to store all row indexes and another (`groupArray`) to store the start index of each group, as illustrated in Figure 4-5b. The row indexes in `rowIndexArray` are ordered in such a way that the row indexes of the first group come first, followed by the row indexes of the second group, and so on. In `groupArray`, we store the index at which each group starts in `rowIndexArray`; that is, `groupArray[0]` contains the index at which the first group starts in `rowIndexArray`, which is 0. The index at which the group ends can be inferred from the start index of the next group or from the size of `rowIndexArray` if there is no next group. Using this structure, we only need to create two arrays and we only need to use 4 bytes to reference the groups



(a) A group storage structure using a list of lists

(b) A group storage structure using two separate lists

Figure 4-5: On the left, we use an array of arrays to store groups. On the right we use one array (`rowIndexArray`) to store the values in all groups, in the order of their groups, and another array (`groupArray`) to store the start indexes of each group in `rowIndexArray`.

instead of 8 bytes.

We will use the second and more efficient structure to build the physical representation for L_{out} in a `group` operator. The block references are the row indexes in `rowIndexArray`, each index represents an index of a DR block in L_{in}, as shown in Figure 4-6b. It is important to note that `getValue()` for the group column returns a list of row indexes with respect to L_{in}, which can then be used to retrieve row values in the group. Below are the `build()` and `getValue()` algorithms.

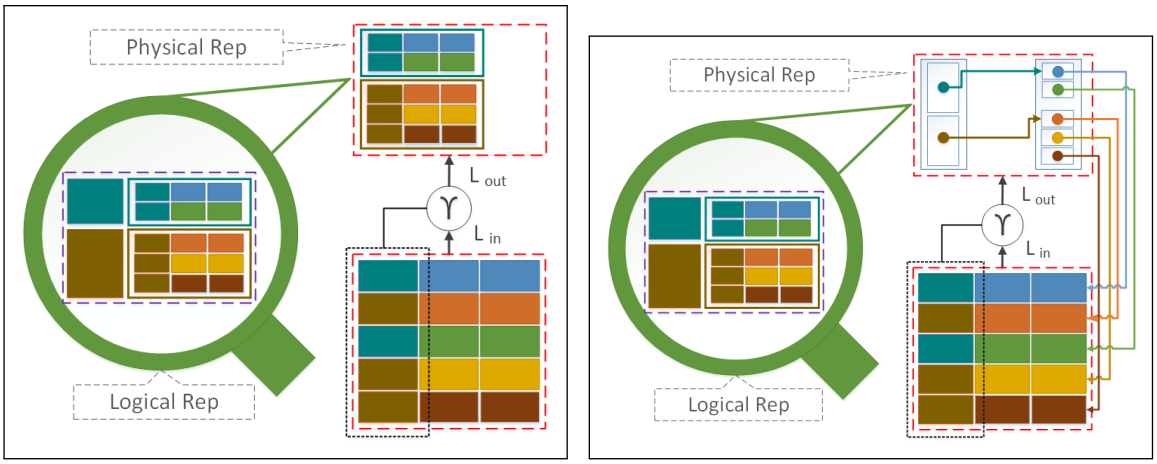
In the `build()` function, lines 2 to 11 constructs the array of groups (`groups`), that is, an array of arrays of row indexes. Lines 13 to 17 constructs an array (`groupArray`) of start indexes of each group in the `rowIndexArray` later. Line 20 merges the groups (the sub arrays in `group`) so that the result (`rowIndexArray`) is an array of row indexes. Lines 22 to 25 constructs the schema of the output layer.

```
1  function build(L_in, groupColList, groupColName)
2    groups = []
3    for i in [0 ... L_in.size()-1]
4      // In groups, find the group that contains rows where the
5      // values of the columns in groupColList match those of
6      // row i.
7      group = findGroup(groups, groupColList, L_in, i)
8      if group == null
9        group = []
10       groups.add(group)
11     group.add(i)
12   // Build the groupArray before we merge the groups.
13   startIndex = 0
14   groupArray = []
15   for i in [0 ... groups.size()-1]
16     groupArray[i] = startIndex
17     startIndex += groups[i].size()
18   // Merge all the sub-arrays in groups in one array, while
19   // maintaining group order.
```

```
20  rowIndexArray = megerSubArrays(groups)
21  // Build the schema
22  schema = []
23  for j in groupColList
24    schema.add(L_in.schema[j])
25  schema.add(new Field{type: collection, name: groupColName})
26  return new GroupLayer{
27    L_in: L_in,
28    schema: schema,
29    data: {
30      groupColList: groupColList,
31      groupArray: groupArray,
32      rowIndexArray: rowIndexArray
33    }
34  }
```

In the `getValue()` function, lines 6 to 8 deal with the case where `j` is one of the grouping columns. If `j` is the group column, we first need to figure out the end index of the group (lines 11 to 14), then we construct a list that contains only the row indexes within the requested group (lines 16 to 18).

```
1  function getValue(i, j)
2    groupStartIndex = this.data.groupArray[i]
3    groupColIndex = this.L_in.schema.size() − 1
4    // Check if j is one of the grouping columns
5    if j < groupColIndex
6      i' = this.data.rowIndexArray[groupStartIndex]
7      j' = this.data.groupColList[j]
8      return this.L_in.getValue(i', j')
9    // To retrieve the value of the group column of group i,
10   // we need the end index of the group
11   if i < this.data.groupArray.size() − 1
12     groupEndIndex = this.data.groupArray[i + 1] − 1
```



(a) A `group` operator that uses replication (b) A `group` operator that uses references

Figure 4-6: A comparison between an `group` operator where data is replicated and a one where data is referenced.

```
13  else
14      groupEndIndex = this.data.rowIndexArray.size() - 1
15  // Build the list of row indexes in group i
16  group = []
17  for i in [groupStartIndex ... groupEndIndex]
18      group.add(this.data.rowIndexArray[i])
19  return group
```

4.1.7 `Aggregate` (Γ)

As shown in Figure 4-7a, the `aggregate` operator aggregates values for a collection of rows based on a list of aggregation functions (`aggFuncList`). In addition to the input layer $\mathtt{L}_{in}$ and `aggFuncList`, the operator also takes as input the collection column (`collCol`) over which the aggregations are performed for each collection (group of rows). The schema in $\mathtt{L}_{out}$ consists of a copy of the schema in $\mathtt{L}_{in}$ plus a column for each aggregation function. Notice that the number of records and their order in $\mathtt{L}_{out}$ is the same as that of $\mathtt{L}_{in}$. If the collection column is not given (`collCol = null`), the aggregations are assumed to be performed over the entire $\mathtt{L}_{in}$ (e.g., count the number

of records in $\mathtt{L}_{in}$). The schema in such a case contains a column for each aggregation function only. Notice that in this case there is only one row in $\mathtt{L}_{out}$. For more on how the `aggregate` operator works, see Section 2.1.4.

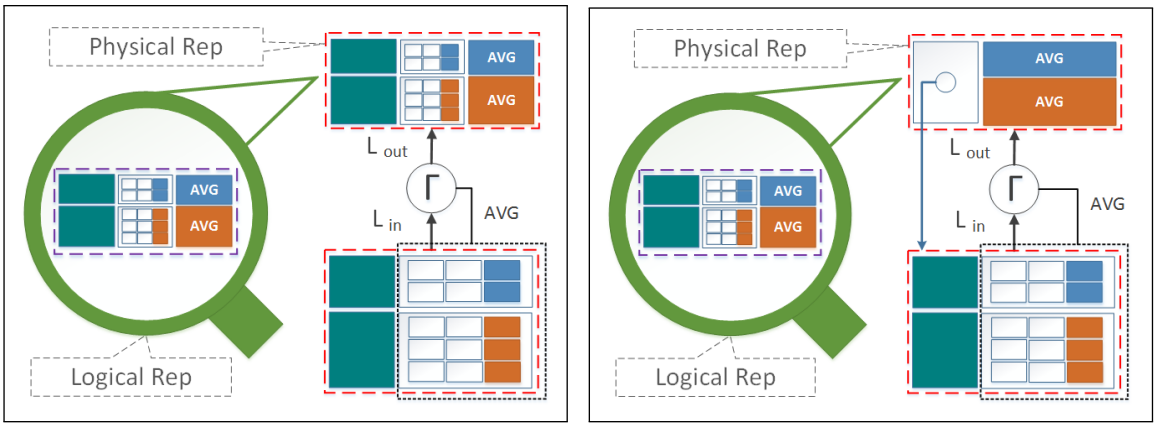
There are two parts to storing the results of an `aggregate` operator. The first part is storing the values of all the columns from $\mathtt{L}_{in}$. Instead of replicating these values (CC, copy column, behavior) in $\mathtt{L}_{out}$, we will use a block reference, RDC (range of data columns), for the entire $\mathtt{L}_{in}$, as shown in Figure 4-7b. The reference is an integer (m) that represents the column index—with respect to $\mathtt{L}_{out}$—before which we need to consult $\mathtt{L}_{out}$ to get the data. In other words, the range of columns to which the block reference refers in $\mathtt{L}_{in}$ is from 0 to $m-1$, which is the entire $\mathtt{L}_{in}$.

The second part that we need to store is the aggregation results. There are two options that we can choose, 1) compute the results on the fly whenever they are accessed and 2) cache the results. If the number of rows in each collection (group) is relatively small, the computations can be done fast and, therefore, the time cost that we save does not justify the space cost that we pay if we cache the results; in such a case, performing the computations on the fly is better than caching. If the number of rows in each collection is large enough so that the total number of rows in $\mathtt{L}_{out}$ is no more than a few thousands, the time cost we save is significant compared to the space cost we need to pay if we cache the aggregation results. In this research, we focus on caching the results rather than computing them on the fly. As future work, the implementation of the `aggregate` operator can be modified so that it analyzes $\mathtt{L}_{in}$ first to determine which option is better, computing on the fly or caching.

Below are the `build()` and `getValue()` algorithms. In the `build()` function, lines 4 to 12 deal with the case where `collCol = null`. In this case, all the rows in $\mathtt{L}_{in}$ are considered as one group and the aggregation functions are applied to that one group; the result (`aggResults`) consists of one row. Lines 14 to 24 deal with the other case where we have a collection column. In this case we collect the result of aggregations for each row in $\mathtt{L}_{in}$. Lines 25 to 29 finishes building the schema based

on the aggregation functions we have.

```
function build(L_in, aggFuncList, collCol)
  aggResults = []
  if collCol == null
    aggColStart = 0
    schema = []
    aggResultRow = []
    for k in [0 ... aggFuncList.size()-1]
      aggFunc = aggFuncList[k]
      // Apply the aggregation function to the entire L_in
      aggResult = aggFunc.apply(L_in)
      aggResultRow.add(aggResult)
    aggResults.add(aggResultRow)
  else
    aggColStart = L_in.schema.size()
    schema = L_in.schema.clone()
    for i in [0 ... L_in.size()-1]
      collectionValue = L_in.getValue(i, collCol)
      aggResultRow = []
      for k in [0 ... aggFuncList.size()-1]
        aggFunc = aggFuncList[k]
        // Apply the aggregation function to the collection
        aggResult = aggFunc.apply(collectionValue)
        aggResultRow.add(aggResult)
      aggResults.add(aggResultRow)
  for k in [0 ... aggFuncList.size()-1]
    aggFunc = aggFuncList[k]
    schema.add(
      new Field{type: aggFunc.returnType, name: aggFunc.alias}
    )
  return new AggregateLayer{
    L_in: L_in,
```



(a) An `aggregate` operator that uses replication

(b) An `aggregate` operator that uses references

Figure 4-7: A comparison between an `aggregate` operator where data is replicated and a one where data is referenced.

```
32      schema: schema,
33      data: {
34        aggColStart: aggColStart,
35        aggResults: aggResults
36      }
37    }
```

In the `getValue()` function, we need to check if the requested column `j` is one of the columns from L_{in} or one of the aggregation values. Line 3 deals with the former, while line 4 and 5 deal with later.

```
1  function getValue(i, j)
2    if j < this.data.aggColStart
3      return this.L_in.getValue(i, j)
4    j' = j − this.data.aggColStart
5    return this.data.aggResults[i][j']
```

4.2 COST ANALYSIS

In the previous section, we discussed space optimizations that we can do to reduce the space cost of intermediate results by using block referencing instead of replicating data. We also discussed the algorithms that build the physical representation of the data layers of each of the seven data operators. In this section we analyze the space and time trade-offs of the space optimizations for each of the six operators (we did not optimize the `import` operator since `import` is just a wrapper).

- The `select` operator: We replace DR blocks with integers. With the exception of rare cases (e.g., the data set has one column with `short` data type), the space cost (SC) of a DR data block is generally much larger than the space cost of an `int` (32 bit). In the vast majority of cases, we replace the larger space cost SC(DR) with a much smaller one SC(`int`) plus the time it takes to dereference `int` ($TC(int)$). In other words, we replace the space cost difference with the dereferencing time cost of calling the `getValue()`. The space cost we save is the space-cost difference between an integer and a DR block times the number of selected rows.

- The `project` operator: We replace an entire column (DC, data column, block) with an integer regardless of the size of the input layer. Not only does `project` cost virtually no space, it also saves significantly on build time (the time it takes to construct the output layer) because no data is actually copied.

- The `union` operator: We save a significant amount of space by replacing the two input layers as a whole with two integers, one each. Not only does `union` cost virtually no space, it also saves significantly on build time because no data is actually copied.

- The `join` operator: Similar to `select`, in `join` we replace two data-row (DR) blocks, one from each input layer, with two integers. The space cost we save is

the space-cost difference (between two integers and two DR blocks) times the number of generated rows.

- The `group` operator: We replace a DR data block with an integer, similar to the `select` operator. The cost we save is the space-cost difference between a data-row (DR) block and an integer times the number of records in the input layer $\mathtt{L}_{in}$. However, we have an extra cost that we need for storing `groupArray`. The size of `groupArray` is less than or equal to the size of `rowIndexArray`; the number of groups is always less than or equal to the number of records that are being grouped. However, in typical group use cases, each group, on average, contains more than one row, which makes the size of `groupArray` at most half the size of `rowIndexArray`. Therefore, the `groupArray` is almost always going to be the dominant space cost.

- The `aggregate` operator: We replace data-column (DC) blocks (the columns transferred from the input layer) with integers, one `int` for each DC block. Although we chose to cache the aggregation results, we still save on space cost because we do not need to store the values for the transferred columns from the input layer. Moreover, aggregations are typically used to reduce the size of data sets to small and manageable sizes that can be inspected manually or be used in visualization tools. In these typical cases, the `aggregate` data layer has a small number of records, which makes the overall space and time costs of caching the results cheap compared to the overall space and time costs of running the aggregations on the fly when the data is needed. By cheap we mean that both the time and space costs stay as far as possible below their defined thresholds. There are cases where aggregations are used for smoothing, in which case the number of records might not be small. In these cases, we can employ a dynamic strategy where the system decides, based on the results and the current space and time costs, whether caching is cheaper than computing the aggregations on

the fly or vice versa.

It is important to note that there are cases where we can do better. For example, in a foreign-key join where the foreign key column does not allow `null` values, we need only one `int` (instead of two) to store the index of the foreign row that matches the foreign key. There are multiple fine-tuning techniques available for special cases for each operator. In this research, we only focus on the main concepts that provide significant space savings. These fine-tuning techniques, although they provide space savings, have marginal savings compared to the main concepts we discuss in this paper.

4.3 SUMMARY

In previous chapters, we argued that keeping intermediate results in main memory is essential for an effective shared data-manipulation system. However, the challenge was the space-cost of keeping those results in main memory. We introduced block referencing as a mechanism to reduce the space-cost of intermediate results by finding data-sharing opportunities across data layers and pointing to the original data blocks instead of copying them. In this chapter, we showed how to implement block references for each operator (excluding `import`) and how to dereference them to acquire the actual data values. Using block referencing provides significant space savings; for some operators there is virtually no space-cost. However, the techniques and the implementations are only valid within the simplified model (only base layers as inputs) that we assumed at the beginning of this chapter. This simplified model is not practical in real data-analysis use cases. In Chapter 5, we will see how to extend the techniques and the implementations from this chapter to support a full model (a full SQL Graph). In addition, we will discuss new techniques to optimize dereferencing time so that we can achieve interactive speed.

CHAPTER 5: TIME OPTIMIZATIONS

In previous chapters, we discussed a shared data-manipulation system where multiple front-end applications can use the system to perform all of their data manipulations, while sharing the results (intermediate or final) with each other. Such a shared system eliminates cross-application data conversion and data movement. However, the biggest challenge to implementing such a system is the space-cost of keeping intermediate results in main memory.

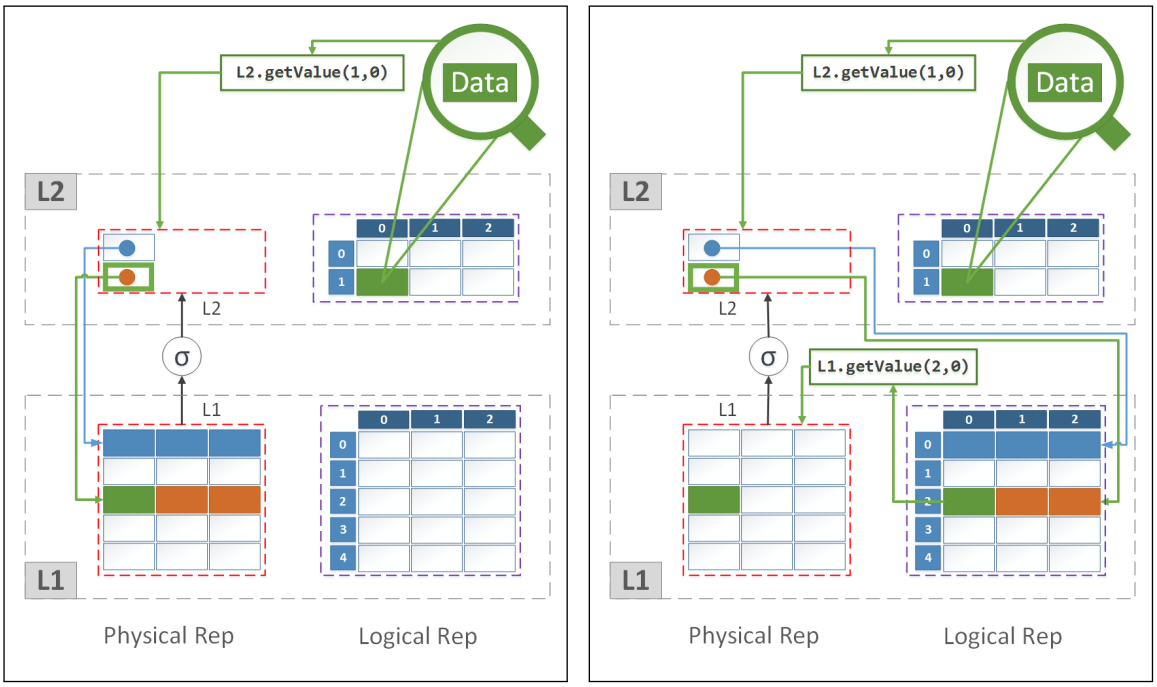
In Chapter 3, we discussed the general mechanism that can help us reduce that prohibitive space cost by using block referencing. However, we stated that for the space-reduction mechanism to be effective, we have to satisfy two conditions: Condition (1) the space cost of a block reference must be less than the space cost of the data block it references and Condition (2) the time cost of dereferencing the block reference to acquire the data must be less than interactive speed. In Chapter 4, we showed how to satisfy Condition (1), but within a simplified model where only base layers are allowed as inputs to the operators. In this chapter, we talk about time optimizations to satisfy Condition (2) but in a full model (a full SQL Graph). We first discuss a naïve approach to extend the techniques we discussed in Chapter 4 to work in a full SQL Graph to allow input layers to the operators be the result of any operator. However, we will see that this naïve approach is expensive in terms of time and does not allow us to satisfy Condition (2). Then, for the rest of the chapter, we discuss time optimizations to satisfy Condition (2).

5.1 BLOCK REFERENCING IN GENERAL SQL GRAPHS

The naïve approach to extending the techniques we described in Chapter 4 to work on a full SQL Graph is to refer to data blocks within the logical representation (logical data blocks) of the input layer instead of referring to data blocks within the physical representation (physical data blocks). The space-saving techniques we described in Chapter 4 assume that the logical and physical representations of the operator's input layers (base layers) are structurally equivalent, as illustrated in Figure 5-1a. This equivalence property no longer holds if input layers can be the result of any operator (a full SQL Graph). However, if we make block references point to data blocks within the logical representation of the input data layers instead of pointing to blocks within the physical representation, we can easily extend the space-saving techniques to a full SQL Graph without changing the physical representations, as illustrated in Figure 5-1b. However, we need to change how we dereference block references to access the data. This naïve approach comes at a high CPU cost to access the data, which we call the *dereferencing cost*.

Since our references now point to logical data blocks relative to the input layers, we have to go through a *dereference-chaining process*, the naïve approach of calling `getValue()` recursively. We have to go through every single data layer along the path starting from the data layer in question all the way to the data layers that contain the physical data blocks, as illustrated in Figure 5-2. As a result, the dereferencing cost is directly proportional to *the height of the data-layer stack* (the number of layers we have to traverse to reach the origin data layer(s)) and, therefore, introduces a linear time complexity (in graph size, not data size) to the dereferencing process. Such linear complexity makes our interactive-speed upper threshold easy to exceed, thus violating Condition (2).

To mitigate the situation, we need to make the dereferencing-cost growth dependent on something that grows more slowly than the height of the data-layer stack,



(a) Block referencing and the dereferencing process in the simplified model.

(b) Block referencing and the dereferencing process in the extended model.

Figure 5-1: In the simplified model (a), we use the physical representation directly. In the extended model (b), we use the physical representation indirectly through the logical representation.

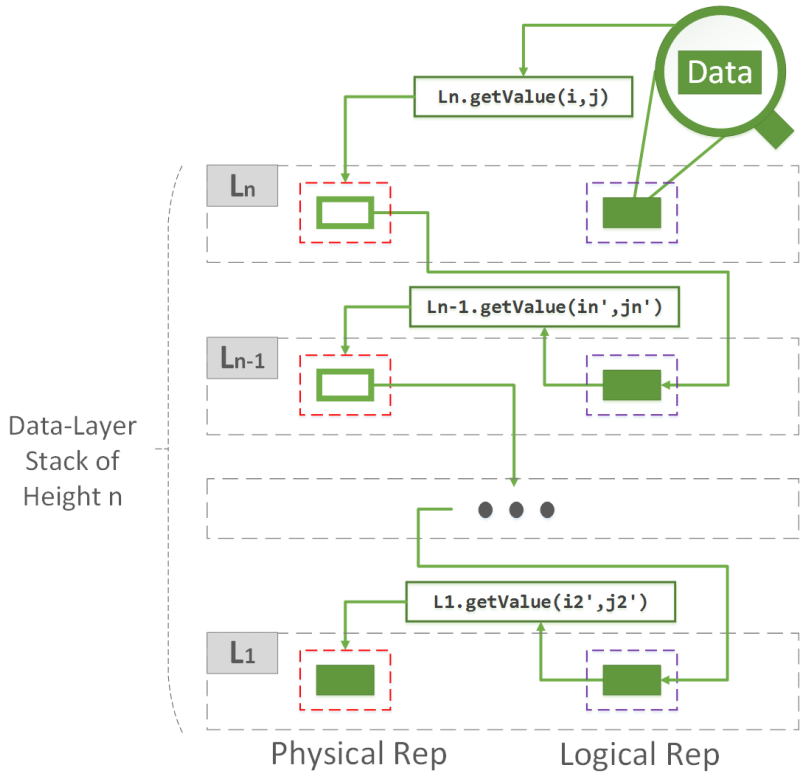


Figure 5-2: The dereference-chaining process in the naïve approach. Data access time depends on the height of the data-layer stack.

at least on average. If we can eliminate time growth for some operators, we improve dereferencing-cost to the extent these operators are used. The rest of this chapter explores how to eliminate time growth for some operators using *eager dereferencing*.

5.2 EAGER DEREFERENCING

The problem that we are facing at this point is that as the height of the data-layer stack grows, the number of steps that we have to go through during the dereference-chaining process also grows. The reason is that each data layer that we add to the stack also adds an extra call to the `getValue()` function. To reduce the number of steps that we have to take during the dereference-chaining process, we will use *eager evaluation*. That is, when we create the data layers, instead of creating block references that point to the input layers, we want to evaluate these references and

make them point to data layers further down the stack, thus skipping steps during the dereference-chaining process. In other words, we pay the dereferencing price once during build time to save us from paying the same price over and over during access time and to prevent that time cost from being passed on to the next layer. We call this eager evaluation of block references *eager dereferencing*.

Ideally, we want at any level in the data-layer stack to eagerly evaluate block references so that they always point to the origin data blocks (the blocks where the actual data resides). That is, we want to call the `getValue()` function just once to reach the data. However, the ideal case is expensive to maintain in terms of space. When we create block references relative to the input layers, data blocks share a significant amount of information (e.g., the input layer), thus we have opportunities to reduce space cost. As the height of the stack increases and the operators that are being applied diversify, shared information becomes scarce, and thus we have to use more space to store block references, to the point where using block references is no better than caching the data. In other words, we reach a point where the space-cost ratio between caching the data and the data-layer's physical representation is close to 1. Later in Section 5.3 we talk about a space efficient way to perform eager dereferencing that allows us to maintain interactive speed. Before we go further, we need to introduce the concepts of *dereferenceable* and *stop-by* layers that will be essential to understanding the eager-dereferencing mechanism later.

5.2.1 Dereferenceable vs. Stop-By Data Layers

However we end up defining the eager-dereferencing mechanism later, we will have operator implementations that produce data layers that can be eagerly dereferenced at build time and others that cannot. We call those data layers that we can eagerly dereference ***dereferenceable*** (DR) layers, and those that we cannot ***stop-by*** (SB) layers. *A DR layer is a layer that can create results that do not require access to the layer*. More precisely, the data layer can convert (or eagerly dereference) its logical

data blocks—which inherently depend on the layer—to equivalent data blocks that depend on layers that are further down the data-layer stack. *An SB layer is a layer that creates results that require access to the layer.* In other words, the dereference-chaining process has to stop by those SB layers to know where to go next to get the data.

We classify our operators' implementations as either DR or SB based on whether the implementation generates a DR or an SB data layer. *An implementation is DR if the operator is able to generate a physical representation* for the output data layer *that can be skipped by the dereference-chaining process. An implementation is SB if it does not generate a DR data layer.* In theory, we can design any operator with redundancy behavior (RUB, Section 3.1) to have DR or SB implementation. For example, we can make all RUB operators have DR implementation by creating a reference for each individual data cell (the ones that were replicated) in the result's logical representation (the result is a data table but with references to the data instead of the data itself). We can also make all RUB operators have SB implementation by materializing the results instead of using block references. However, neither end of the spectrum is generally[1] space efficient; creating a reference for each data cell in the result requires more or less the same amount of space as materializing the result.

In Chapter 4, all of our operators have SB implementations, since they create block references that depend on the input layer(s). The goal is to have as many operators as possible with DR implementations while maintaining an overall small space footprint. We next talk about a space-efficient technique that we call the *dereferencing layout index* (DLI) that supports DR implementations for many of our operators by replacing the operator's generated physical representation with a DLI referencing structure.

[1]There are rare cases where, for example, materialization is efficient. For example, if we perform a `select` on a layer with one column whose data type is `byte`, materializing the data is more efficient than using references. However, for `union`, using references is still far more efficient.

5.3 THE DEREFERENCING LAYOUT INDEX (DLI)

To recap, we want to be able to dereference $\mathtt{L}_{in}$'s block references as we apply the operator to build $\mathtt{L}_{out}$ so that the block references at $\mathtt{L}_{out}$ do not point to $\mathtt{L}_{in}$, but rather point to a layer further down the data layer stack. In other words, if the block references in $\mathtt{L}_{in}$ point to layer $\mathtt{L}$, when we build $\mathtt{L}_{out}$, the block references in $\mathtt{L}_{out}$ should continue to point to $\mathtt{L}$ instead of pointing to $\mathtt{L}_{in}$. The problem with the space-saving techniques we discussed in Chapter 4 is that they rely on implicit assumptions depending on the input layer(s) and the operator that generates the output layer. If any of these assumptions changes, the operators' dereferencing algorithms (`getValue()`) become invalid. For example, `project`'s algorithm (Section 4.1) relies on the assumption that the block references point to the immediate underlying layer. If we want to skip that layer and instead reference a layer further down the data-layer stack, we cannot guarantee that row i at $\mathtt{L}_{out}$ corresponds to row i at $\mathtt{L}_{in}$ (Figure 4-2).

We need a mapping mechanism to tell us that a given row i and a column j at layer L correspond to i' and j' at layer L'. We propose a space-efficient mapping data structure we call a ***dereferencing layout index*** (DLI). Similar to block referencing, a DLI maps blocks of data instead of individual cells; the bigger the blocks, the fewer entries we need in the DLI, the less the DLI costs in terms of space. We refer to such a mapping block as a ***dereferencing layout*** (DL). In other words, a DLI provides bulk dereferencing instead of cell-level dereferencing, thus sharing space and dereferencing costs. *A DLI of a layer L is a set of DLs where each data cell in the logical representation of L is covered by exactly one DL.* The assumption is that all data cells that are covered by a given DL come from the same layer L' and that their i' and j' can be computed using $i' = f(i)$ and $j' = g(j)$, where f and g are mappings that can be represented as an array or an expression. Note that for the implementations of the operators that we discuss in this chapter, we only use arrays or the identity function for f and g. However, other operator implementations might

use other expressions for f and g (see Section 8.1.1 for an example).

5.3.1 Dereferencing Layout (DL)

In general, DLs provide a bulk-mapping mechanism that can mutate and adapt based on the operator that we apply. The DLs rely on a principle that we call the ***property-sharing assumption***, which states that *the data cells that a given DL covers share certain properties that we can use for dereferencing to obtain the data-cell values*. The design of DLs determines which operator implementations are DR and which ones are SB. *If we cannot find property-sharing opportunities for an operator, we will need an SB implementation.* It is important to note that all implementations for the `import` operator are inherently SB and, therefore, all base layers are SB. The reason is because a base layer is where the original data is stored; there is no next step in the dereferencing process.

We designed the DLs so that each DL maps a range of rows in a layer L at once. We found that choosing a range of rows results in far more operators having DR implementations than with a range of columns. For example, with range of rows, we can have DR implementations for the operators `select`, `project`, `union`, `distinct`, `aggregate-ref`, and `semi-join` (we discuss some of these implementations in Section 5.3.4). On the other hand, with range of columns, we can have DR implementations for the operators `join` and `project`. The intuition is that a range of whatever must all come from the same reference layer. If an operator breaks that condition, we can no longer carry that DL to the next layer, and we have to stop by this operator's layer to know where to go next. Since there are more operators that manipulate rows than columns, it makes sense that we get more operators with DR implementation using a range of rows than a range of columns.

Our design of a DL maintains the values of the following five shared properties:

1. SRI: The start row index.
2. L': A reference layer.

3. f: A row mapping.

4. g: A column mapping.

5. ERI: The end row index.

The idea is that to resolve a given row i and a column j at layer L, find the DL in L's DLI such that $SRI \leq i \leq ERI$, then call L'`.getValue(`i'`,`j'`)`, where L' is an SB layer, $i' = f(i - SRI)$, and $j' = g(j)$. In other words, a DL tells us that for a given row $SRI \leq i \leq ERI$ and a column j, the next stop in the dereference-chaining process is the SB layer L', where the correct i' and j' to use at L' are $f(i - SRI)$ and $g(j)$, respectively. The reason we use $i - SRI$ in f instead of just i is because the row-index references in the row map are zero-based relative to L', which allows us to share f by reference across data layers. The definitions of the functions f and g vary based on the data layer's type. We will define f and g precisely for each operator in a bit, but for now, you can think of f and g as arrays of indexes.

We can look at the behavior and the implementation of our operators and determine which ones break the *property-sharing assumption* (the rows no longer share the same five properties once they propagate to the output layer) and which ones do not. The implementations that do not break the property-sharing assumption are DR implementations and the ones that do are SB implementations. Based on our design of DLs, an operator's implementation can break the property-sharing assumption in two ways:

- Since our design of DLs assumes that a range of row indexes share the five properties mentioned above, any implementation that is not row-order preserving (ROP) can cause rows to switch DLs as they propagate to the next layer, thus invalidating the assumption. For example, say we have two DLs DL_1 and DL_2 that cover row indexes 0 to 5 and 6 to 10 in L_{in}, respectively. If the operator's implementation is not ROP, row 2, for example, in L_{in} might become row 7 in L_{out}. However, row 7 is covered by DL_2, which assumes that all rows from

indexes 6 to 10 get their data from the data layer $DL_2.L'$, which in $\mathtt{L}_{out}$ is not true because the data for row 7 come from the data layer $DL_1.L'$.

- The operator generates a new column. Note that we are assuming that the implementation caches the results at the resulting data layer. Since our design of a DL is about a range of rows and assumes that all columns for that range of rows come from the same reference layer, adding a new column invalidates that assumption.

Out of the seven operators we discuss in this research, we were able to come up with DR implementations for `select`, `project`, and `union`. Chapter 8 talks briefly about other operators for which we produced DR implementations as well. Since the `import` operator is a wrapper to a working data data, it inherently has an SB implementation because that is where the data originates. The operators that are left to have SB implementations are `join`, `group`, and `aggregate`. Note that for some of the columns (the aggregations themselves) in an `aggregate` data layer, the dereference-chaining process stops at the `aggregate` layer because that is where the data originates. In other words, by using an `aggregate` operator, we reduce the average dereferencing cost[2], sometimes even reset the dereferencing cost to zero[3]. So not all SB implementations necessarily mean an increase in dereferencing cost.

For the rest of this section, we discuss how to integrate DLIs into the framework that we established in Chapter 4. Then we discuss the SB and DR implementations within our definition for a DL. Then we discuss the general dereferencing algorithm, the `getValue()` function, for all DR implementations. In the subsequent section, we go over an example to show how these concepts work together and how DLIs help reduce the dereferencing cost.

[2]The average dereferencing cost is the average time cost to dereference a data reference across all the needed columns. Note that for some columns the data is materialized at the `aggregate` layer, whereas for others the data is not, so we have to continue the dereferencing process.

[3]The dereferencing cost resets to zero when all the needed data at the `aggregate` layer come from the materialized columns

5.3.2 The Integration of DLIs

In Chapter 4, we stated that a data layer has three main attributes, the input layer(s), the schema, and the physical representation (`data`). In addition to the three attributes, now we add a new attribute called `DLI`. In addition to the function `build()`, we also add a new function `buildDLI()` to each operator class for the operator to build its DLI. (We talk about how in a bit.) Operators with SB implementations keep their `build()` implementations as we discussed in Section 4.1 in addition to calling `buildDLI()` to set their `DLI` attribute. On the other hand, DLIs now become the physical representation (the content of the `data` attribute) of operators with DR implementations. Next we talk about the implementation of the `buildDLI()` function.

5.3.3 Operators With SB Implementations

All operators with SB implementations produce what we call the ***unit DLI***. A unit DLI, as illustrated in Figure 5-3, is a DLI that contains one DL with the following property values:

1. SRI points to the first row (0) in L (the layer that the SB implementation generated and that contains the unit DLI).
2. L' points to L itself.
3. f and g are the identity functions.
4. ERI points to the last row ($L.size() - 1$) in L.

The following is the implementation for `buildDLI()` function in all operators with SB implementations:

```
function buildDLI(L)
  DL' = new DL{L': L, SRI: 0, f: (i) -> {return i},
    g: (j) -> {return j}, ERI = L.size() - 1}
  DLI_out = [ DL' ]
```

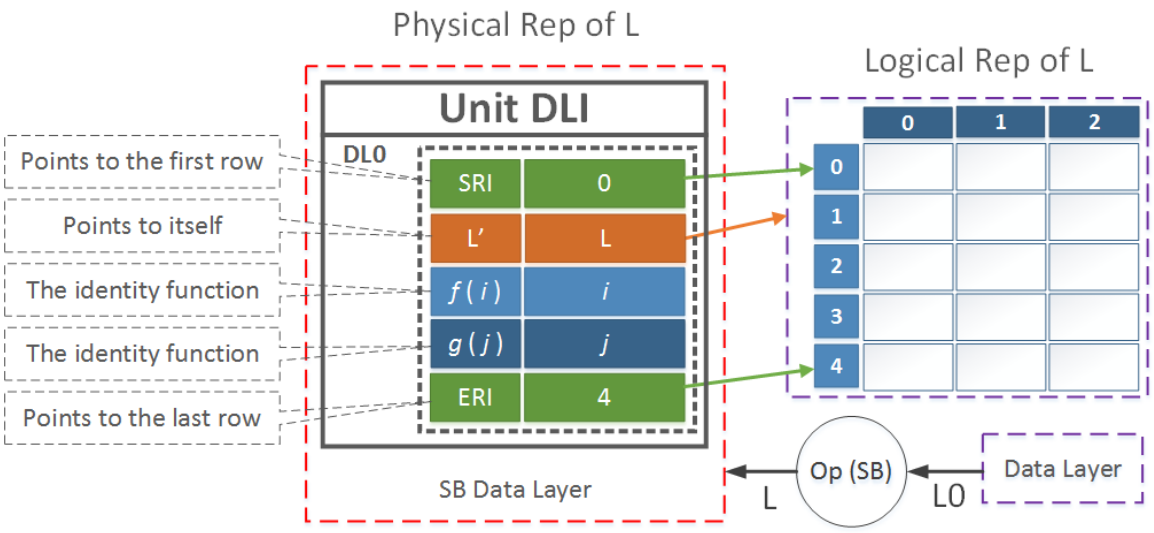


Figure 5-3: Creating a unit DLI by SB implementation.

```
5  return DLI_out
```

The idea of a unit DLI is to serve as a building block for other DLIs. As we apply operators with DR implementations to L, the DL inside the unit DLI propagates to the next layer with some amendments to its properties while maintaining its reference to L (the SB layer). In later layers, any row that is between the SRI and the ERI of this propagating DL requires a stop by L to acquire its data values.

Notice that the unit DLI in an SB layer is not used for dereferencing, it is strictly used as a building block to build other DLIs in DR layers above it. Otherwise, the dereferencing process will never halt once it reaches an SB layer. In other words, when we ask an SB layer for a data value at row i and a column j, we simply use the dereferencing algorithms (the `getValue()` function) described in Section 4.1 instead of using the unit DLI.

5.3.4 Operators With DR Implementations

The general idea for any DR operator is that the DLI now becomes the primary physical representation that the operators produce and becomes the general dereferencing

mechanism that any DR layer uses to provide its logical representation. We discuss first how each operator with a DR implementation builds its DLI, then we discuss the general dereferencing algorithm (`getValue()`) that all DR layers use to provide their logical representation.

Generally, every operator with a DR implementation uses the DLI(s) of its input layer(s) ($\texttt{DLI}_{in}$) as building blocks for the operator's output DLI ($\texttt{DLI}_{out}$). The behavior of the operator's implementation determines how the input layer's DLs should be amended and added, if at all, to $\texttt{DLI}_{out}$. The following describes how each of our DR-implementation operators builds and amends its DLI.

5.3.4.1 `Select`

If the operator selects 100% of the rows in $\texttt{L}_{in}$, $\texttt{L}_{out}$ should have an exact replica of $\texttt{L}_{in}$'s DLI ($\texttt{DLI}_{in}$). We need to modify $\texttt{DLI}_{in}$ once a DL loses a row because of the `select` predicate. The general idea to create $\texttt{L}_{out}$'s DLI ($\texttt{DLI}_{out}$) is that for each DL_{in} in $\texttt{DLI}_{in}$, we want to create DL_{out} in $\texttt{DLI}_{out}$ such that it covers only the rows in $\texttt{L}_{in}$ that are covered by DL_{in} and that satisfy the predicate. In Figure 5-4, we see how the DLI of $\texttt{L}_{in}$ (`L3`) should be amended to create the DLI of $\texttt{L}_{out}$ (`L4`). There are three cases that we need to cover for each DL_{in} in $\texttt{DLI}_{in}$:

1. Some (but not all) of the rows that are covered by DL_{in} do not satisfy the predicate, as illustrated in Figure 5-4 in $\texttt{DL}_0$ (in `L3`'s DLI) where row 0 does not satisfy the predicate. In such a case, we need to create a new row map from `L4` to `L1` for the f function to have only the rows that satisfy the predicate. In addition, we also have to update `SRI` and `ERI` to reflect the new row-range coverage with respect to `L4`.

2. All the rows that are covered by DL_{in} satisfy the predicate, as illustrated in Figure 5-4 in $\texttt{DL}_1$ (in `L3`'s DLI). In such a case, the row map from `L4` to `L2` for

the f function is the same as that of the row map from `L3` to `L2`. Instead of creating a new row map, we can simply copy-by-reference the one we already have in $\texttt{DL}_1$ from `L3` to save space. The only difference however, is that we have to update `SRI` and `ERI` to reflect the new row-range coverage with respect to `L4`.

3. None of the rows that are covered by DL_{in} satisfy the predicate. In such a case, we do not need to create a corresponding DL_{out} and we can simply ignore DL_{in} altogether.

Note that because `select` does not change the schema of $\texttt{L}_{in}$, the column maps for the g functions are exactly the same in all cases. So we can simply copy-by-reference the g functions in all the DLs to save space. The following is the algorithm that the `select` operator uses to build $\texttt{DLI}_{out}$, given $\texttt{DLI}_{in}$ and the selection `predicate`:

```
1  function buildDLI(DLI_in, predicate)
2    DLI_out = [ ]
3    startRow = 0
4    for DL in DLI_in
5      RM = [ ]
6      for i in [DL.SRI ... DL.ERI]
7        i' = DL.f(i − DL.SRI)
8        if predicate(DL.L', i')
9          RM.add(i')
10     if RM.size() > 0
11       if RM.size() == (DL.ERI − DL.SRI + 1)
12         f' = DL.f
13       else
14         f' = (i) -> {return RM[i]}
15       DL' = new DL{L': DL.L', SRI: startRow, f: f',
16         g: DL.g,  ERI: startRow + RM.size() − 1 }
17       DLI_out.add(DL')
18       startRow = DL'.ERI + 1
19   return DLI_out
```

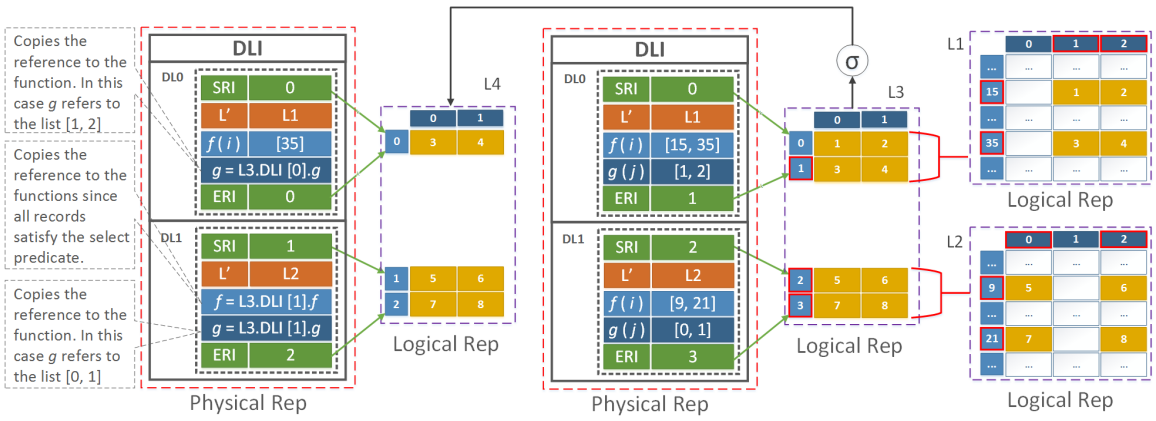


Figure 5-4: An illustration of how a `select` operator changes the DLI of the input layer `L3` to produce the DLI of the output layer `L4`. To the right, we see the original data layers from which the data blocks in `L3` come. As we apply `select` to `L3` to select rows 1, 2, and 3, we see how the individual DLs in `L3`'s DLI are modified to reflect the new status of `L4`'s logical data blocks.

5.3.4.2 `Project`

The way we alter the DLI for the `project` operator is similar to the `select` operator. The only difference is that we update the column maps for the g functions instead of the row maps for the f functions. If the operator retains 100% of the columns in the same order (although such use defeats the purpose of using `project` in the first place), $\texttt{L}_{out}$ should also have an exact replica of $\texttt{L}_{in}$'s DLI ($\texttt{DLI}_{in}$). We need to modify $\texttt{DLI}_{in}$ if the operator rearranges or drops one or more columns. What we want in such a case is that for each DL_{in} in $\texttt{DLI}_{in}$, we need to produce DL_{out} for $\texttt{DLI}_{out}$ such that it covers only the projected columns. Specifically, we want to update the column map (g function) of DL_{in} to reflect the new column mapping from $\texttt{L}_{out}$ to $DL_{in}.\texttt{L}'$. Since `project` does not affect rows, all the row maps (f functions) that are passed on from $\texttt{DLI}_{in}$ are still valid and, therefore, we can copy them by reference to save space.

Compared to `project` in Section 4.1, there is an extra space cost due to the g function in each DL. However, typically the number of columns is small and, therefore, the space cost of the column maps (g functions) is cheap compared to the CPU gain we get from using DLIs when we use it for dereferencing. The following is the algorithm

that the `project` operator uses to build $\texttt{DLI}_{out}$, given $\texttt{DLI}_{in}$ and a list of projected column indexes (`colIndexList`):

```
function buildDLI(DLI_in, colIndexList)
  DLI_out = [ ]
  for DL in DLI_in
    CM = [ ]
    for j in colIndexList
      CM.add(DL.g(j))
    DL' = new DL{ L': DL.L', SRI: DL.SRI, f: DL.f,
      g: (j) -> {return CM[j]}, ERI: DL.ERI }
    DLI_out.add(DL')
  return DLI_out
```

5.3.4.3 `Union`

The `union` operator simply takes both inputs' DLIs and merges them into one. Specifically, we want to take the DLs of $\texttt{L}_{in2}$'s DLI ($\texttt{DLI}_{in2}$) and append them to the DLs of $\texttt{L}_{in1}$'s DLI ($\texttt{DLI}_{in1}$). However, we need to offset the row coverage of each DL from $\texttt{DLI}_{in2}$ by the number of rows covered by $\texttt{DLI}_{in1}$. In other words, if $\texttt{DLI}_{in1}$ and $\texttt{DLI}_{in2}$ cover m and n rows, respectively, in their own input layers, the same DLIs will cover in $\texttt{L}_{out}$ the rows 0 to $m-1$ and m to $m+n-1$, respectively. So for every DL_{in} in $\texttt{DLI}_{in2}$, we need to create DL_{out} and add m to `SRI` and the `ERI`.

The use of DLIs in `union` incurs an extra space cost compared to the implementation in Section 4.1. However, all DLs in $\texttt{DLI}_{in1}$ are copied by reference, no new DLs are created. For $\texttt{DLI}_{in2}$, although we create new DLs, we copy f and g (the dominant space-cost factor) by reference. As long as we maintain a small number of DLs in any given DLI, the overall space cost of a `union` DLI ($\texttt{DLI}_{out}$) is negligible. The following is the algorithm that the `union` operator uses to build $\texttt{DLI}_{out}$ given $\texttt{DLI}_{in1}$ and $\texttt{DLI}_{in2}$:

```
function buildDLI(DLI_in1, DLI_in2)
  DLI_out = [ ]
```

```
startRow = 0
for DL in DLI_in1
    DLI_out.add(DL)
    startRow = DL.ERI + 1
for DL in DLI_in2
    DL' = new DL{ L': DL.L', SRI: startRow + DL.SRI,
        f: DL.f, g: DL.g, ERI: startRow + DL.ERI }
    DLI_out.add(DL')
return DLI_out
```

5.3.4.4 Other Operators

Later, in Chapter 8, we briefly mention other operators for which we were able to come up with a DR implementation and, therefore, able to use DLIs. However, there are operators for which we might not be able to come up with a DR implementation. As we mentioned in Section 5.3.1, how we design DLs determine whether an operator's implementation can be DR or SB, except the `import` operator whose implementations are all inherently SB. The design that we chose for DLs is about a range of rows sharing certain properties. One of those properties is that all the rows in a given range come from the same reference layer (L'). So any operator implementation that creates rows whose columns come from different layers, such as `join`, is not DR and, therefore, we cannot use DLIs for the output layer. Similarly, any implementation that adds a new column to the output layer, such as `group` and `aggregate`, also creates rows with columns that map to different reference layers[4]. Note that the standard `project` operator also creates new columns (computed columns) in addition to propagating existing columns from the input layer. However, as we describe in Chapter 8, we were able to modify the design of DLs to map columns to expressions, which allowed the standard `project` operator to have a DR implementation.

The last issue that prevents us from finding a DR implementation (regardless

[4]The newly added column(s) come from the output layer, while the other columns do not.

of the chosen design for the DLs) for an operator is materializing results. Any operator implementation that materializes results, such as our implementation for the `aggregate` operator, is inherently SB. However, unlike other types of implementations that generate SB layers, these result-materializing implementations reset the dereferencing cost back to zero. That is, once the dereference-chaining process reaches a materialized SB layer, the process ends. So although this kind of SB layers is inefficient in terms of space, it is very efficient in terms of time.

5.3.5 DLI Dereferencing Algorithm

Although each operator has its own implementation of how to amend and update the input DLI(s) based on the operator and the behavior of its implementation, all operators with DR implementation have the same dereferencing algorithm (the `getValue()` function). Because of DLIs, the dereference process can skip all DR layers in a given stack down to an SB layer, thanks to the eager dereferencing that we perform at build time. We discuss an example in the next section to see how. So the dereference-chaining process only hops from one SB layer to the other. We start by calling the `getValue()` on the data layer in question (the data layer from which the user wants to retreive data) given a row i and a column j. Since the DLs in a given DLI are kept ordered by SRI, we can use a binary search to find the DL that covers a given row i (`findDLContains()`). Once we find the DL, `L'` tells us the SB layer that we need to visit next (call `L'.getValue()`), but we should use $f(i)$ and $g(j)$ instead of i and j. The process continues until we reach origin data layers (e.g., an `import` layer). The general dereferencing algorithm (`getValue()`) for all DR layers is as follows:

```
1 function getValue(i, j)
2   DL = findDLContains(this.DLI, i)
3   i' = DL.f(i − DL.SRI)
4   j' = DL.g(j)
5   return DL.L'.getValue(i', j')
```

5.3.6 DLI Example

Figure 5-5 shows an example that applys a `select` operator followed by a `project` followed by a `union`. Formally the query for the layers that we want to create is:

$\texttt{L5} = \cup\ (\pi_{1,2}\ (\sigma_{row\_index\ \ in\ \ (0,2,3)}\ (\texttt{L1})),\ \texttt{L4})$.

The query starts with `L1`, which is an SB layer and, thus, it has a unit DLI. Then we apply a `select` to `L1` to select rows with indexes 0, 2, and 3 and build layer `L2`, as shown in Figure 5-5 part A. To build `L2`'s DLI ($\texttt{DLI}_{out}$), we use `L1`'s DLI ($\texttt{DLI}_{in}$). Following the `select`'s `buildDLI()` algorithm, we go through each DL in $\texttt{DLI}_{in}$ and see which case we need to apply. $\texttt{DLI}_{in}$ has only `DL0`, which follows Case 1 since some of the rows (1 and 4) covered by the DL do not satisfy the predicate. In that case, we need to create a new `DL0` in $\texttt{DLI}_{out}$ such that:

- $\texttt{DLI}_{out}.\texttt{DL0}.SRI = \texttt{DLI}_{in}.\texttt{DL0}.SRI$, which is 0.
- $\texttt{DLI}_{out}.\texttt{DL0}.L' = \texttt{DLI}_{in}.\texttt{DL0}.L'$, which is `L1`.
- $\texttt{DLI}_{out}.\texttt{DL0}.f =$ `[0, 2, 3]`. That is, rows 0, 1, and 2 at `L2` come from rows 0, 2, and 3, respectively, at `L1`.
- $\texttt{DLI}_{out}.\texttt{DL0}.g = \texttt{DLI}_{in}.\texttt{DL0}.g$, which is the identity function. The function is copied by reference.
- $\texttt{DLI}_{out}.\texttt{DL0}.ERI = 2$, only 3 out of five rows from `L1` satisfied the predicate. These rows start at index 0 and end at index 2.

Next we apply a `project` to `L2` to select columns 1 and 2 to build layer `L3`, as shown in Figure 5-5 part B. To build `L3`'s DLI ($\texttt{DLI}_{out}$), we use `L2`'s DLI ($\texttt{DLI}_{in}$). Following the `project`'s `buildDLI()` algorithm, we go through each DL in $\texttt{DLI}_{in}$ and update the column maps to reflect the new mapping from `L3` to `L1`. There is only `DL0` in $\texttt{DLI}_{in}$, so we create a new `DL0` in $\texttt{DLI}_{out}$ such that:

- $\texttt{DLI}_{out}.\texttt{DL0}.SRI = \texttt{DLI}_{in}.\texttt{DL0}.SRI$, which is 0.
- $\texttt{DLI}_{out}.\texttt{DL0}.L' = \texttt{DLI}_{in}.\texttt{DL0}.L'$, which is `L1`.
- $\texttt{DLI}_{out}.\texttt{DL0}.f = \texttt{DLI}_{in}.\texttt{DL0}.f$, which is `[0, 2, 3]`. The function is copied by reference.
- $\texttt{DLI}_{out}.\texttt{DL0}.g =$ `[1, 2]`. That is, columns 0 and 1 at `L3` come from columns 1 and 2, respectively, at `L1`.
- $\texttt{DLI}_{out}.\texttt{DL0}.ERI = \texttt{DLI}_{in}.\texttt{DL0}.ERI$, which is 2.

Now assume that we want to get the value for the data cell at row 2 column 1 at `L3`. We see that row 2 is covered by `DL0`, which says, based on the value of `L'`, that the next stop is `L1`. The correct i and j to use at `L1` are $i = \texttt{DL0}.f(2 - \texttt{DL0}.SRI)$, which is 3, and $j = \texttt{DL0}.g(1)$, which is 2. Notice that we completely skipped `L2`.

The final step is to apply a `union` to `L2` and `L4` to build layer `L5`, as shown in Figure 5-5 part C. To build `L5`'s DLI ($\texttt{DLI}_{out}$), we use `L3`'s DLI ($\texttt{DLI}_{in1}$) and `L4`'s DLI ($\texttt{DLI}_{in2}$). Following the `union`'s `buildDLI()` algorithm, we merge both inputs' DLIs and update the SRI and ERI of $\texttt{DLI}_{in2}$'s DLs. Since both inputs' DLIs have one DL, $\texttt{DLI}_{out}$ will have two DLs, `DL0` and `DL1`, such that:

- $\texttt{DLI}_{out}.\texttt{DL0} = \texttt{DLI}_{in1}.\texttt{DL0}$. That is, we copied `DL0` by reference from $\texttt{DLI}_{in1}$.
- $\texttt{DLI}_{out}.\texttt{DL1}.SRI = \texttt{DLI}_{in2}.\texttt{DL1}.SRI + 3$. The number 3 is the number of rows in `L3`.
- $\texttt{DLI}_{out}.\texttt{DL1}.f = \texttt{DLI}_{in2}.\texttt{DL1}.f$, which is the identity function. The function is copied by reference.
- $\texttt{DLI}_{out}.\texttt{DL1}.g = \texttt{DLI}_{in2}.\texttt{DL1}.g$, which is the identity function. The function is copied by reference.
- $\texttt{DLI}_{out}.\texttt{DL1}.ERI = \texttt{DLI}_{in2}.\texttt{DL1}.ERI + 3$. Again, 3 is the number of rows in `L3`.

Assume that we want to get the value for the data cell at row 2 column 1 at `L5`. We see that row 2 is covered by `DL0`, which says, based on the value of `L'`, that the next stop is `L1`. The correct i and j to use at `L1` are $i =$ `DL0`$.f(2)$, which is 3, and $j =$ `DL0`$.g(1)$, which is 2. Notice that we completely skipped `L3` and `L2`. If we want to get the value for the data cell at row 5 column 1 at `L5`, this time we use `DL1`, which points to `L4` as the next stop where $i =$ `DL1`$.f(5 -$ `DL1`$.SRI)$, which is 2, and $j =$ `DL1`$.g(1)$, which is 1.

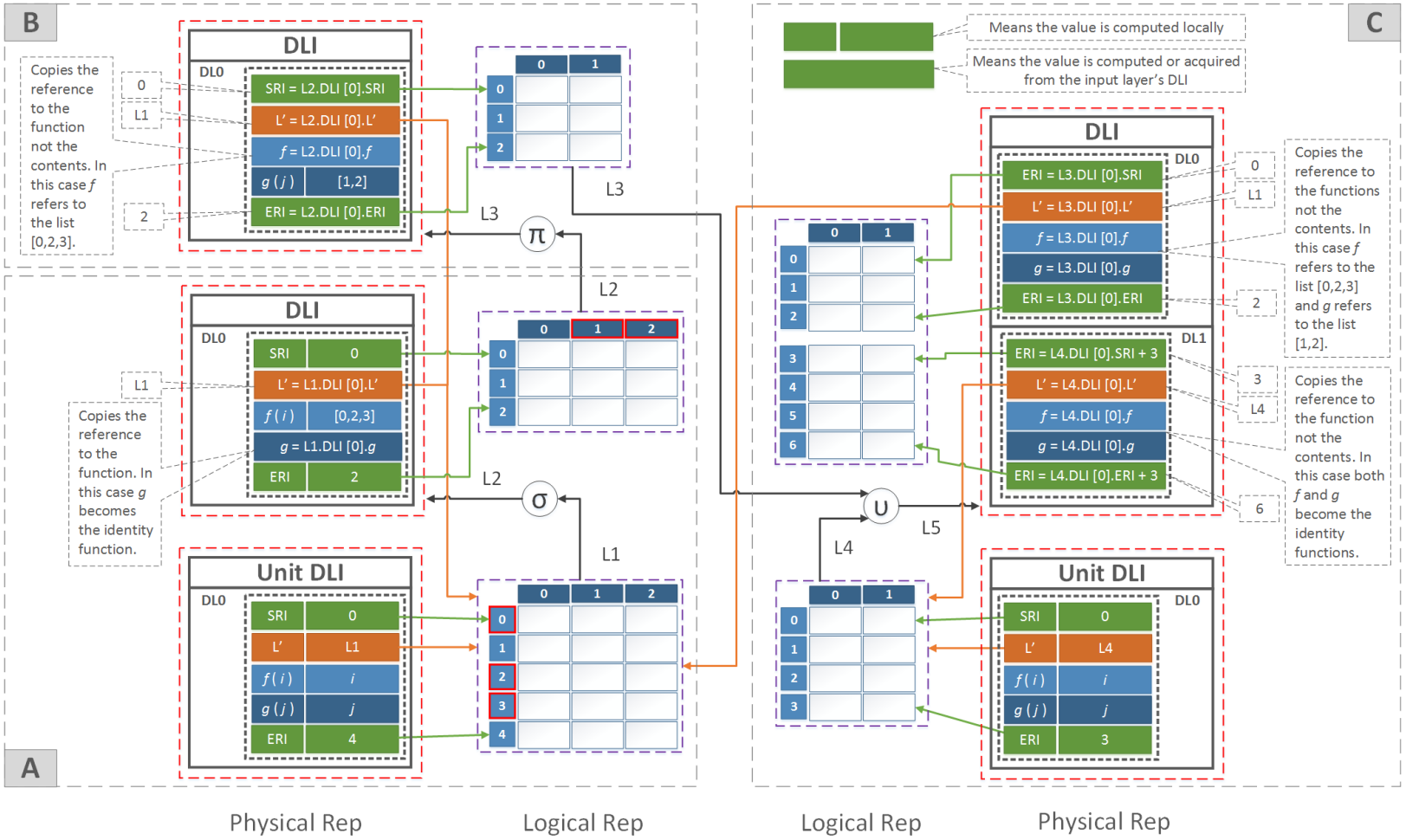


Figure 5-5: An illustration of how DR operators create DLIs using the input layer's DLI(s). In **A**, the σ operator takes an SB layer (`L1`) with a unit DLI to produce `L2`. In **B**, the π operator takes a DR layer (`L2`) and uses its DLI to produce `L3`. In **C**, the $\cup$ operator takes a DR layer (`L3`) and an SB layer (`L4`) and uses their DLIs to produce `L5`.

5.4 COST ANALYSIS

The techniques in Chapter 4 provide significant space savings to the operators' intermediate results. However, these techniques are only valid when input layers are base layers. In Section 5.1, we naïvely extended these techniques to work on a full SQL Graph, but at a fast-growing dereferencing cost, which prevents us from expanding SQL Graphs to practical-data-analysis sizes without violating Condition (2). We introduced DLIs and DR implementations for some operators to slow down the dereferencing-cost growth by skipping DR layers during the dereferencing process. The claim is that this deceleration is enough to allow SQL Graphs to expand large enough for typical data-analysis use cases to take place before we violate Condition (2) or run out of memory. The more operators with DR implementations we have, the slower the dereferencing cost grows on average. However, the effectiveness of DLIs relies on maintaining a small space footprint.

There are three factors that dominate a DLI's space cost: the number of DLs it contains and the size of the f and g functions in each DL. The number of DLs becomes a major factor only if the size of the data-layer stack becomes exceptionally and unrealistically large (e.g., > 10000). The g function also becomes a major factor only if the number of columns at a given layer is exceptionally large (e.g., > 1000) and we have to create a new column-map list (as opposed to copying one by reference, which costs virtually nothing). So the only concerning factor is the size of the f function.

For a unit DLI, the size of f costs virtually nothing since it is the identity function[5]. Therefore, we did not add any extra cost to SB layers beyond what we discussed in Section 4.2. For DR layers, DLIs either added a negligible space cost or made extra space savings in exchange for effective time savings. In `select`, the f function is still a list of integers that reference DR blocks. However, if all the rows that are covered

[5]The space cost of an identity function is very small and fixed regardless of the size of the data set to which it refers.

by a given DL are selected, we can point to the original DL's f function instead of creating a new one, which saves more space. In `project`, all f functions are copied by reference, which costs virtually nothing. In `union`, all DLs from the first input layer are copied by reference. For the second input layer, although we create new DLs, their f and g functions are copied by reference. The bottom line is, the addition of DLIs added a negligible space cost to the cost of the techniques used in Chapter 4, while allowing us to extend these techniques to a full SQL Graph with a slow, growing dereferencing cost, as we will demonstrate in Chapter 7.

To understand the experiments and the results that we discuss in Chapter 7, we first need to discuss the in-memory storage that we used to store working data sets. So the next chapter (Chapter 6) discusses two approaches that we used and their advantages and disadvantages.

CHAPTER 6: IN-MEMORY STORAGE ENGINE

Up to this point, we have only focused on techniques to reduce the size of intermediate results and to improve data-access time. These techniques provide large space savings and allow us to extend the SQL Graph significantly with limited memory (RAM) space. Although storing intermediate results is the biggest space-cost factor (which is the focus of this research), reducing the space cost of working data sets and the materialized data in data layers can further extend the SQL Graph, especially when the working data sets are large. For example, if we start with 6 GB of available memory and our working data set costs 4 GB to store in memory, we only have 2 GB left for storing intermediate results to analyze the data. Suppose we are able to store the results of 60 operators using the techniques that we have discussed in the previous chapters. If we reduce the size of the working data set even by half, we can potentially double the amount of intermediate results that we can store to 120 operators.

In this chapter, we first discuss a naïve approach that we used initially to store the working data sets and the materialized data (such as in the `aggregate` operator). We also discuss the consequences of using such a naïve approach on space and access time. Then for the rest of this chapter we discuss a space-efficient way that we used to store the data and that enabled us to execute realistic data-analysis use cases, which we discuss in Chapter 7.

Note that what we discuss in this chapter is not necessarily a novel data-storage approach nor is it intended to be. The intent of this chapter is to provide a complete picture for our research and discuss some of the challenges and the consequences that we faced as a result of using such naïve data-storage approaches. We also believe

that this chapter provides important context to the experiments and results that we discuss in Chapter 7.

6.1 THE NAÏVE DATA-STORAGE APPROACH USING JAVA OBJECTS

In the first prototype we built, we were mainly concerned about testing the space-saving techniques for storing intermediate results. Since the efficiency of storing working data sets is not the focus of our research, we used built-in Java data structures such as `HashTable`s and `ArrayList`s. Java data structures are general-purpose abstractions and they work with Objects instead of primitive data types. As a result, there is a significant amount of unnecessary overhead in terms of space and time that is being added behind the scenes. Even when we used Java's built-in `ByteBuffer` class to store the data for a given row, the space overhead that each `ByteBuffer` object creates is still too big for our purposes.

Another issue with using Java (or any JVM language for that matter) is that the JVM does not return unused memory to the OS. Moreover, the JVM will not know that a given space in memory is unused until the garbage collector (GC) runs. So if a certain computation creates a large number of temporary objects or creates temporary objects with a large footprint, these objects consume memory very quickly even if the final result that remains in memory is small. Even though the total amount of data that is being retained might be small, the application itself could still claim a significant amount memory.

Using `ArrayList`s causes significant wasted space and time, especially during computations. Since an `ArrayList` is supposed to give the illusion of a dynamic array (in terms of size), the abstraction has to keep creating new arrays with the appropriate sizes behind the scenes whenever the array becomes full and move the contents to the new arrays. This process creates a significant amount of unused space that seems to be difficult to utilize later even after the GC runs, because of memory fragmentation.

We used `HashTable`s as temporary data structures to store computation intermedi-

ate results, such as performing hash joins and grouping results in `group` and `distinct` operators. Using `HashTable`s turned out to be the Achilles' heel for space efficiency. For each entry in the `HashTable`, we need to add a key and a value for that key. Even if the key and the value are integers, Java still represents those two integers as Objects. In addition to the relatively large size of each object (as opposed to the size of an integer), we also need two object pointers (8 bytes each) for the key and the value. Moreover, the implementation of a Hash Table in Java or any other language makes the whole process very expensive in terms of space. For example, there is always the question of what the initial size should be. Choosing a large size (w.r.t. the size of the data) will almost certainly result in wasted space, while choosing a small size will most certainly hurt time performance because of the high probability of collisions. Moreover, any technique for handling collisions in Hash Tables results in extra space overhead. For example, if we use Linked Lists, we need an object for each node and we need extra pointers to link the nodes. Without using Hash Tables, we can no longer use certain techniques (at least not in a traditional way) such as hash joins, and we have to settle for slightly less time-efficient but far more space-efficient techniques, such as sort-merge joins.

This naïve way of storing data, whether it is the materialized results or during computations, results in a space-consumption explosion, especially when the data is large. In addition, the creation of all of these unnecessary objects that happens behind the scenes adds extra time during both build and access time. With such performance overhead (in space and time), it was not possible for us to use the prototype that we created to test realistic use cases with large data sets and with large SQL Graphs. So it was important for us to spend some time optimizing our operator's core algorithms to use customized space-efficient data structures during computations. The biggest space saving we obtained (after data-block referencing) is the customized data-storage engine for working data sets and for materialized data resulting from some operators (e.g., the `aggregate` operator). The rest of this chapter describes the structure of this

new customized data-storage engine.

6.2 THE CUSTOMIZED DATA-STORAGE ENGINE

As we mentioned earlier, the storage engine that we discuss in this section is an in-memory storage engine and is designed to store data efficiently (for space and time) for working data sets and for materialized data resulting from some data operators. We set two criteria to achieve with our design for the storage engine. The first criterion is that we want to eliminate as much wasted space as we possibly can, which means we need a compact way to arrange the data. The second criterion is that we want data-access time to be as low as possible. We cannot use compression because all general-purpose compression algorithms require decompression to access the data (see Section 9.5 for more on compression algorithms), which is expensive. There are in-memory compression techniques [2, 10, 14, 24, 48, 68] to reduce the decompression cost, such as caching, but for now we want to reduce the space cost without adding any noticeable access-time cost. In future work, we will experiment with in-memory compression algorithms to see if the space-savings justify the access-time overhead from compressing and decompressing the data.

Many of the design choices of our storage engine are influenced by common row-store structures in many database management systems [55, Chapter 9]. Figure 6-1 shows the general storage structure of a given data set. The data set is divided into pages, each of which is a byte array. Each page has three segments. The first segment, **data storage**, is where the data itself is stored. The second segment is the **Null bitmap**, where we store the information to determine whether a given field in a given record is `null` or not. The last segment is the **row-position map**, where we store information to tell us where each record starts and ends in the data-storage segment. Next we discuss each of these three segments in detail.

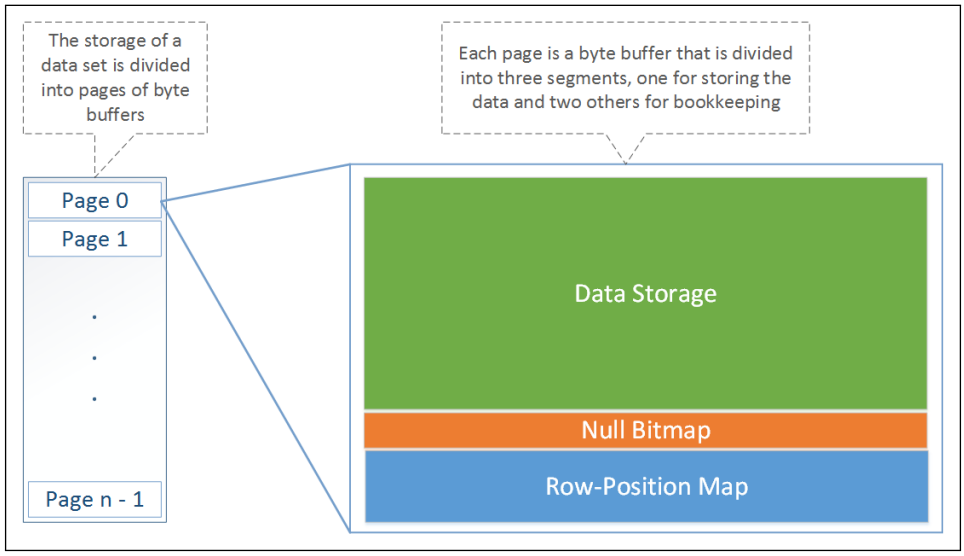


Figure 6-1: The general structure of a data storage of a data set. The data storage is divided into a list of pages, each of which is a byte array. Each page is divided into three segments. The start location of the Data Storage segment is the beginning of the byte buffer. The end location of the Data Storage segment, which is also the beginning location of the Null Bitmap segment, is stored in the last four bytes of the byte buffer. The start location of the Row-Position Map segment can be calculated using the equation $(BufferSize - 4 - NumOfRowsInPage * 2)$.

6.2.1 The Data-Storage Segment

In the data-storage segment of the page we store the actual data. Figure 6-2 shows the internal structure of this segment. The data is arranged one record after the other. The first record is at the beginning of the segment, while the last record in the page is at the end of the segment. For each data record, the data is serialized (converted into a stream of bytes) and arranged so that the fixed-length fields (e.g., `int`, `boolean`, and `double`) come first, then followed by the variable-length fields (e.g., `String`). For the variable-length fields, we first place the references (byte positions) for the starting position of each of the variable-length fields that has data. Each reference is 4 bytes. (We can reduce the reference size to less than 4 bytes with more careful design, but we need to test the affect of such a design on data-access time, because it will add extra overhead when we compute the field offset.) After the references, we place the values for the variable-length fields.

For any given record, we do not store any information in this segment if a field's value is `null`. For example, if the schema has three fields and, for a given record, the value for the second field is `null`, the data will be arranged as the first-field value followed by the third-field value, and nothing in between. Whenever we access the data, we first check the Null flag in the Null-bitmap segment (which we discuss in a bit). If the field's value is flagged as Null, then we return `null` as the value. Otherwise, we compute the field value's offset based on the number of Null fields that precede the field in question.

The maximum size of this segment depends on the data, however, there is one important rule. The byte offset (the position in the byte array) of the beginning of the last record must not be more than $2^{16} - 1$ or 65535. So if we exceed that number in a given page, we have to start another page. The reason for this rule is so that we can record the offset of the beginning of each record using only two bytes, which saves space in the third segment of the page.

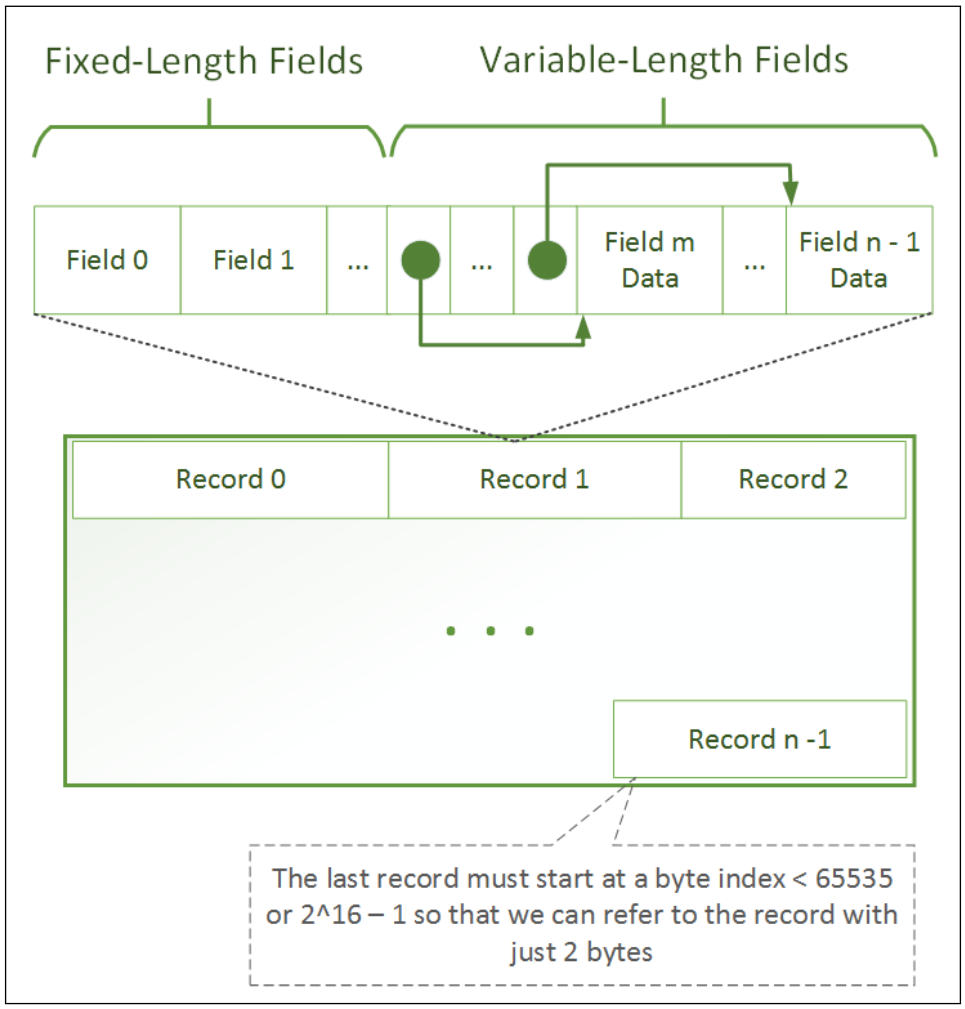


Figure 6-2: The internal structure of the Data Storage segment in a page. The data is arranged by rows. For each row, the data for the fixed-length fields precede the variable-length fields.

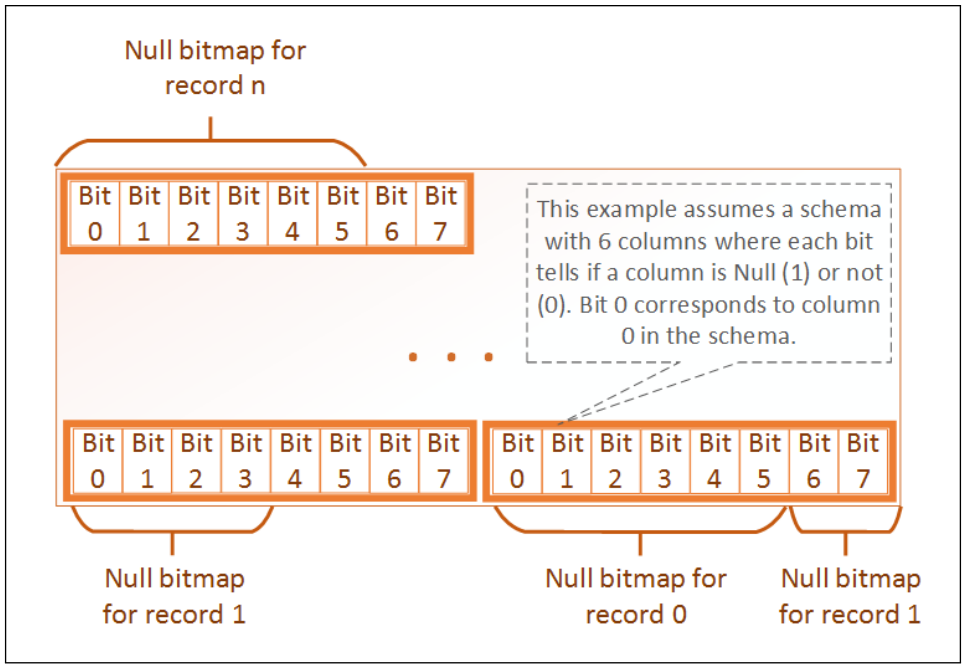


Figure 6-3: The internal structure of the Null bitmap segment in a page. There is a bit for each column for each row in the page. The bits are arranged from the bottom-up. That is, the Null bitmap information for the first row is in the last byte, whereas the last row contains the last row's info.

6.2.2 The Null-Bitmap Segment

An important aspect about storing data is to be able to distinguish between a valid data and no data (`null`) for a given field. In our storage-engine design, we chose to use a dedicated segment for Null bitmaps as opposed to storing the Null bitmap inline with the data records themselves. This design choice saves extra space because we can fully utilize every byte of the Null-bitmap segment (except perhaps the last byte). Figure 6-3 shows how we store the Null-bitmap information in this segment. Unlike the previous segment, the Null-bitmap for the first record starts at the end of the segment and continues backwards. This arrangement saves a few instructions when we access the data, because it is compatible with the actual byte order (big-endian). So bit 0 in the last byte in the Null-bitmap segment corresponds to the first field in the first record in the page.

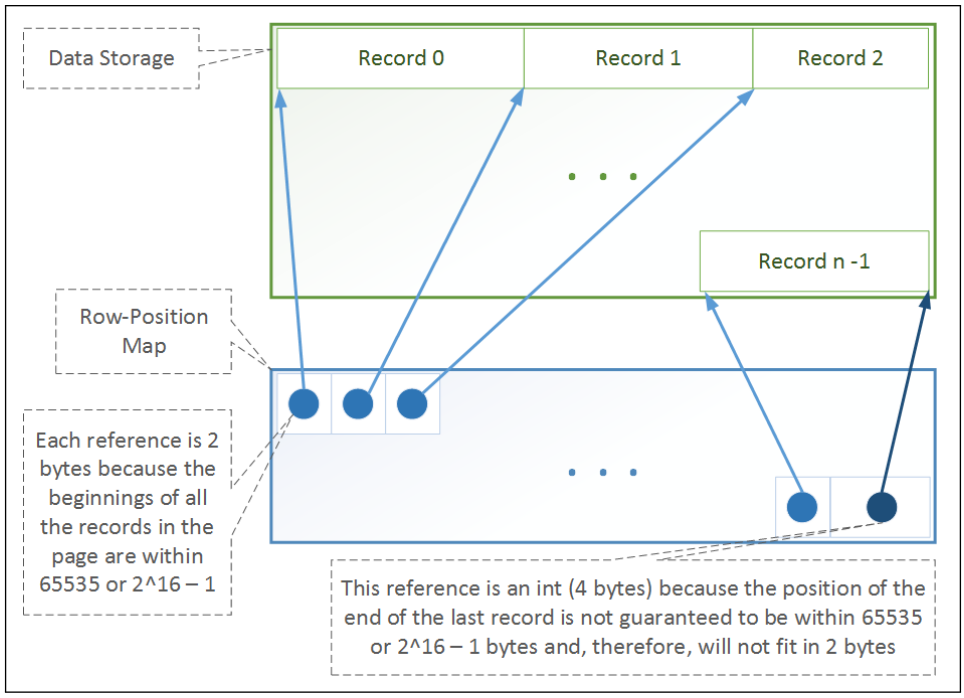


Figure 6-4: The internal structure of the row-position map segment in a page. This segment contains information about where each record in the data-storage segment starts. The position information (byte offset) for each record is stored in in two bytes (short). In addition, the last four bytes contain the position where the data in the data-storage segment ends.

6.2.3 The Row-Position-Map Segment

The last segment in the data-storage page is the row-position map. Since we arranged all the data rows in one stream of bytes, we need a way to determine where each row starts and ends as illustrated in Figure 6-4. We can capture the offset (the byte position) of the beginning of each row as we add rows to the page. Because we require that each row must start at a byte offset less than or equals to $2^{16} - 1$ or 65535, we can store the byte-offset information using only two bytes (`short`).

To determine the end of a given row, we can simply use the offset of the next record. However, this approach does not work for the last record in the page. So the last 4 bytes in each page are used to store the end offset of the data-storage segment or the end offset of the last record in the page. The reason why we need 4 bytes is

because we cannot guarantee that the last record always ends at an offset less than or equals to 65535. Although we can determine the end of the data-storage segment by other means and without consuming extra 4 bytes, such an approach would cost extra instructions during data-access time. So we believe that 4 bytes per page is a negligible space cost that saves valuable data-access time.

6.3 DISCUSSION

This chapter is not intended to provide a comprehensive explanation for our improved data-storage engine. As we mentioned at the beginning of this chapter, storing working data sets is not the focus of this research. However, optimizing our data storage engine was necessary for us to be able to evaluate realistic use cases on large data sets. It is worth noting that trying to access data using just what we described in this chapter is time consuming. For example, we need a way to index the pages to know in which page each record is. Since we do not store `null` values, we also need a fast way to compute the offsets of each field within a given row, given the Null bitmap of that row. There are complementary data structures (not discussed in this chapter or in this research) that we have used to solve these issues and speed up the data-access process. The extra space cost of these data structures is negligible compared to the overall space cost of the data storage itself. For example, a data set that costs 1 GB would need only a few hundred kilobytes for the extra data structures.

Using this new storage engine, we were able to achieve space savings more than three quarters of the space we need using the naïve-storage approach. Moreover, the application's memory consumption becomes more stable and much more predictable. We were also able to achieve a slight improvement in data access time. Although computing record and field offsets in theory costs more time than using the naïve approach using Java objects, in practice, we also eliminated a significant amount of unnecessary, behind-the-scenes overhead caused by the general-purpose nature of Java data structures. In Chapter 7, we show a comparison between the naïve storage

engine and the newly improved one.

CHAPTER 7: EXPERIMENTS AND RESULTS

The goal is to build a shared data-manipulation system in a client-based environment. We discussed the importance of keeping the data and the intermediate results in main memory. Up to this point, we introduced the concepts and the techniques that should allow the shared data-manipulation system to keep intermediate results in main memory in a space-efficient way (SQL Graphs) to handle long data-analysis sessions. In this chapter, we test these claims with three experiments and show how far we can extend the SQL Graphs in client-based environments.

The first experiment (Section 7.2) is a synthetic use-case that is designed to eliminate or greatly reduce the effect of factors that are unrelated to this research. This use-case allows us to accurately measure the effectiveness of the techniques that we have discussed. The main questions we want to answer with this experiment are:

1.1. How effective are the space-saving techniques compared to materialization?

1.2. How effective are DLIs in reducing the dereferencing cost?

The second experiment (Section 7.3) tests the new customized storage engine that we discussed in Chapter 6 and compares it to the old naïve implementation. The main questions we want to answer with this experiment are:

2.1. How much space do we save using the new storage compared to the old one?

2.2. How efficient is data-access time using the new storage compared to the old one?

The last experiment (Section 7.4) is a realistic data-analysis use-case that we used for a class project in the past. In this experiment we repeat the analysis process but,

this time, we use our system prototype and three other systems instead of the original system that we used for the class project. The main question we want to answer with this experiment is:

3.1. How does our system prototype compare to other known, well developed systems in terms of space cost, build time, and access time?

For each experiment, we will talk about the use-case and the experimental setup, then discuss the results, and finally briefly discuss what we learned from the experiment.

Before we start, a quick note on access time versus build time.

Definition 7.1 (**Access time**)**.** Access time is the time it takes for an algorithm to acquire the data from its storage.

Definition 7.2 (**Build time**)**.** Build time is the time it takes to run the algorithm to completion, which includes the time it takes to access its input data.

In other words, access time is how long it takes to retrieve the values from its storage, and build time is access time to the input values plus whatever we need to do with that value.

7.1 THE ENVIRONMENT SETUP

In all of the experiments we used a desktop PC with Intel i5, 3.50GHz CPU with four cores (though all of our operations are single-threaded) and an 8GB RAM. The OS is Linux Ubuntu 18.04.5 LTS 64-bit. We used Java version 1.8 for all the experiments that require Java. In the third experiment (Section 7.4) we used three other systems in addition to our prototype: PostgreSQL [28] version 9.5. Spark [7] version 2.4.5, and MySQL [18] version 5.7.

To test the techniques that we have discussed in this research, we built a prototype for a shared data-manipulation system that we call the *jSQL environment* or jSQL_e for short. We wrote jSQL_e in Java, hence the "j" part of the name. We also created

a new query language for our system that we call jSQL. The language is designed to be an imperative language instead of the typical declarative SQL language. The imperative aspect of the language makes it suitable for the exploratory-analysis data model for which the system is designed. We tried to make the language as close as possible to the standard SQL to make it familiar and easy to learn for those who already know SQL. You can see examples of jSQL in Appendix A.4. The system however, evolved throughout the three experiments. With the first experiment, the system was still basic, where the operators used basic and naïve algorithms to process the data and we also used the naïve storage engine. However, the techniques that we discussed in this research, up to and including Chapter 5, were fully implemented. For the second experiment, we added the new, customized storage engine that we discussed in Chapter 6. For the final experiment, we spent three months optimizing the core algorithms of our data operators and adding a query optimizer, so that jSQL_e can have a fair comparison with other systems. Note that the optimizations that we added exist in almost every database management system. For example, there are many algorithms that can be used to perform a join, each of which is suitable for certain situations. The trick is to figure out at runtime based on the given parameters which algorithm to use. For such a job, database management systems have query optimizers.

7.2 SYNTHETIC USE-CASE

In this experiment we want to test how far we can extend the SQL Graph, in terms of data-layer stack height and the overall number of data layers, before we violate Condition (2) or run out of memory. The questions we want to answer are:

1. How effective are the space-saving techniques compared to materialization?
2. How effective are DLIs in reducing the dereferencing cost?

We constructed an unrealistic use-case to push the limits of the concepts that we

introduce in this research and see how far we can go. The goal is to extend the SQL Graph to be large enough to support usual sizes of typical data-analysis use-cases. We define a typical size of a typical use-case as a data-analysis session with over a hundred operators and stacks of data layers as high as 20. We also do not focus on build time (Definition 7.2) of data layers, only access time (Definition 7.1) of values in these layers. The reason is that build time is a combination of access time and the time it takes to run the operator's core algorithm[1], which is not the focus of this research.

7.2.1 Experiment Setup

The environment setup is as discussed in Section 7.1. In this experiment, we used the naïve storage engine to store the working data sets. When we did this experiment, we had not yet implemented the new, customized storage engine. The data set that we used had about 4.3GB of in-memory footprint, eight columns, and about 30 million records, which is on the large side[2] for a client-based data analysis. The data we used is real data that was collected from highway detectors that collect volume, speed, and occupancy with a timestamp. In addition, we also have metadata for detectors with 100KB of in-memory footprint, 15 columns, and 444 records.

The goal is to build a stack of data layers and measure the cost growth of space and time as we add more data layers. We add more layers by performing a *split-merge process*: we take the top layer and split it into two halves using two `select` operators then combine both results back again using a `union` operator. The reason we do the split-merge process is to prevent other factors from contributing to both costs (positively or negatively) by keeping the original number of records and schema when we test for both time and space. To measure the accurate space footprint of

[1]The operator's core algorithm refers to the part of the operator's algorithm that processes the data, such as how a `join` joins two rows or how a `select` filters a row once the data is acquired.

[2]Given the limited capabilities of client-based machines (typically desktops and laptops with about 8GB of RAM), a data set with millions of records can push the limits of many systems, such as spreadsheets, R, and many visualization tools

each operator's result, we run the JVM garbage collector (GC) after we execute each operator and then we measure the amount of memory used by the JVM instance. To test for time and CPU cost, we run the *min-max query* "find min and max timestamp" at the stack's top layer and measure how long it takes to finish. The query scans the top layer sequentially and touches every record.

We tested three types of stacks:

1. An SB stack (SB), where all operators have SB implementations (Chapter 4).

2. A DR stack (DR), where operators have DR implementations using DLIs (Chapter 5).

3. A DR stack with join (DR w. $\bowtie$). This stack type is the same as DR stack but we added a `join` (with SB implementation) in the middle of the stack. The `join` is simply a foreign-key join with the detectors-metadata data set. The join keeps the original number of records, while the schema expands. Using `join` allows us to see the impact of SB implementation on the dereferencing cost.

In all three types, we start with the baseline Stack 0 where the only layers in the SQL Graph are the base layers. Then we generate Stack i by performing the split-merge process on Stack $i-1$; we test ten stacks ($i \in [0, 10]$) in total. Notice that the height of each stack is $2i + 1$, because each split-merge adds two levels, one for the two `select`s and the other for the `union`. Figure 7-1 illustrates the process of building the stacks.

Although the upper limit for interactive speed is typically between 500ms and 1sec (depending on the application), for our test we use 2sec as the interactive-speed threshold. The reason is that the time it takes to execute the query at the base layer for 30m records is a little above 1sec. In addition, the focus of the experiment is the growth in time (and space) rather than the initial cost. We also set the main-memory space limit to 6GB for the application to use for storing intermediate results or otherwise, leaving 2GB for the OS.

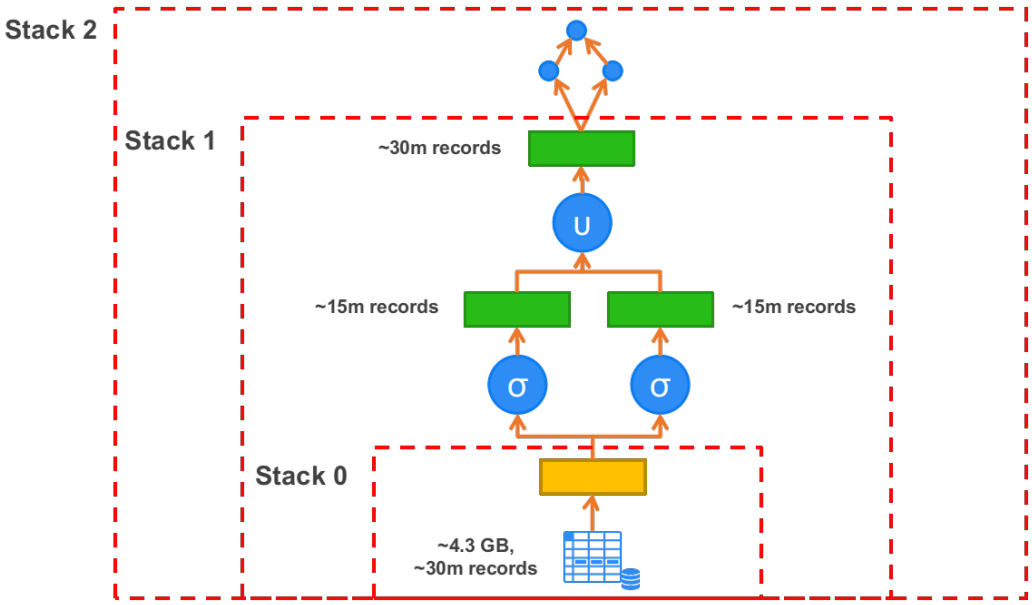
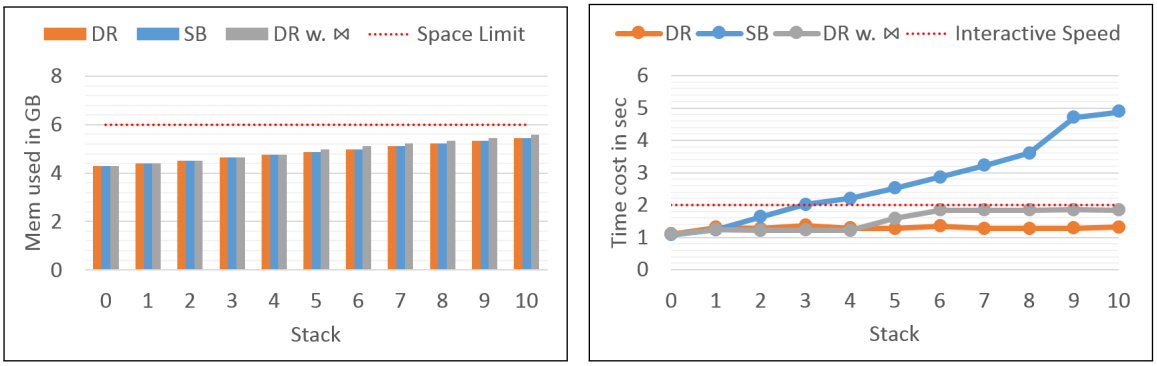


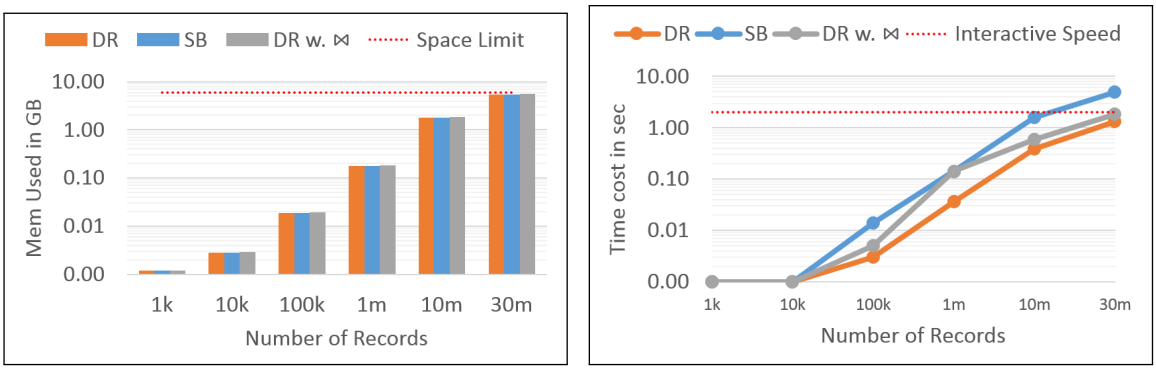
Figure 7-1: Building the stacks for the synthetic use-case. Stack 0 consists of only the base layer. Stack 1 builds on top of Stack 0 by adding three layers (two `select` and one `union`), which increase the height of the stack by two (the two `select` layers are on the same level). Stack 2 builds on top of Stack 1 using the same previous process. We repeat the process until we reach Stack 10.



(a) Space cost

(b) Access time cost

Figure 7-2: The space and access-time costs as the stack grows in size for an initial data set with 30m records.



(a) Space cost

(b) Access-time cost

Figure 7-3: The space and access-time costs of Stack 10 as the number of records in the initial data set grows. Note that the y-axes are on logarithmic scales, and the x-axes increases geometrically except for the final entry.

7.2.2 Results

Figure 7-2 shows the results for all ten stacks in addition to the base Stack 0, where the base data layer has 30m records. Figure 7-2a shows the overall memory used after constructing each stack, whereas Figure 7-2b shows the time it takes to execute the min-max query at the top layer at each stack. Keep in mind that Stack 0 has only the base layers (two[3] layers), while each subsequent stack adds three more layers (two `select`s and one `union`) to the previous stack and increases the height of the data-layer stack by two levels (both `select`s make up the first level, while `union` makes up the second). In *DR stack with join*, the `join` layer replaces the two `select` and the `union` operators in Stack 5.

In Figure 7-2a, Stack 0 shows the original data-set's space footprint, which can be used to calculate how much it would cost, space-wise, to materialize the data at the subsequent data layers. For example, in Stack 1 we add two `select` layers (∼15m records each) and a `union` layer (∼30m records). If we were to materialize the data in each of the three layers, we would need an extra 8.6GB to store the data. By

[3]The SQL Graph for the experiment starts by applying the `import` operator on each of the two working data sets that we have, the highway data and the detectors' metadata. The result of each `import` operator is a base layer, thus we have two base layers to start with.

Stack 10, we would need a total of 90.3GB to keep the entire SQL Graph in memory. However, by using block referencing, we only needed an extra 116MB for the added three layers. By Stack 10, we only needed 5.5GB to keep the entire SQL Graph in memory, including the initial data set (∼4.3GB). Since block references in `join` require more space (232.7MB), we see a bump in memory usage from Stack 5 to 10. On the other hand, there is virtually no difference between an SB and a DR stack in terms of memory usage, which means that the use of DLIs did not add any extra space cost. Notice that by Stack 10, we were able to keep the intermediate results of 30 operators (28 in the case of *DR stack with join*), each result with an average of 20m records, without running out of memory. Also notice that, unlike materialization, the space cost of block referencing is not affected by the data set's schema size; only the number of records and the type of the data layer drive the space cost.

In Figure 7-2b, Stack 0 shows the time it takes to run the min-max query. It also shows the time it takes to run the query if we were to materialize the data at the top layer of each stack. Moreover, Stack 0 gives us the baseline that we want to stay as close to as possible without crossing the interactive-speed line. As we add more layers in subsequent stacks, the time cost grows linearly in the *SB stack* and, by Stack 3, we exceed the interactive-speed threshold. On the other hand, the *DR stack* maintains a constant time all the way to Stack 10. The *DR stack with join* also maintains a constant time until Stack 5 where the `join` is added, then grows again at Stack 6 and continues to be constant after that. The reason for the second increase is that all DR layers after the `join` must make two stops to reach the data blocks, one at the `join` and another at the immediate underlying layers.

In Figure 7-2, we tested extreme cases where the average number of records per layer is around 20m. To get a sense of the cost of using block referencing in more realistic use-cases, we ran the same experiment again, but we reduced the number of records in the original data set to 10m, 1m, 100k, 10k, and 1k. Figure 7-3 shows the time- and space-cost comparison among these data-set-size variations at Stack 10.

Figure 7-3a shows the space footprint of the three types of tests, while Figure 7-3b shows the time it takes to run the min-max query at the top layer. Notice that the y-axes are on logarithmic scales. Table 7.1 shows the growth of space and time cost at Stack 10 with respect to Stack 0 as the number of records in the original data set increases.

From Table 7.1 and Figure 7-3, we can calculate how far we can extend the SQL Graph before we run out of memory or exceed interactive speed. For example, if we start with 1m records and construct a DR stack, the total memory used at Stack 10 is ∼180MB with a 38MB growth from Stack 0, an addition of 3.8MB per stack. To reach the space limit that we defined (6GB), we need to extend the SQL Graph to $\sim$ 1531 stacks $((6GB - 180MB)/3.8MB)$ or the equivalent of 4593 data layers $((2\sigma + 1\cup) * 1531)$ with an average of ∼667k records per data layer. Although the access-time cost grew 1ms from Stack 0, the overall cost (36ms) stayed constant throughout all 10 stacks. The constant time cost means, in theory, a DR stack can grow indefinitely and will never exceed the interactive-speed limit (2sec). The 4593 layers can be all in one stack or can be spread across multiple stacks (e.g., exploring different data analysis paths).

For an SB stack, the time-cost growth at Stack 10 is 103ms from the time cost at Stack 0 (143ms), an addition of 10.3ms per stack. Although we can still extend the SQL Graph to contain 4593 data layers, we can have only 540 data layers $((2sec - 143ms)/10.3 \approx 180$ stacks, or $(2\sigma + 1\cup) * 180$ layers) in any given stack to stay under the time threshold. The last thing we want to mention is the time-cost growth of the *DR with join* stack. The addition of a single `join` caused an increase of 12ms over the time-cost growth of a pure DR stack. This increase means we can have no more than ∼166 join layers (2000/12) in any given stack. Note that we are talking about the foreign-key `join` that we used in this experiment, which maintained the number of records (30m records) from the input layer. In realistic use-cases, the number of records might increase or decrease as a result of applying a `join`. If the number

of records decreases, the space and access-time cost decrease, and if the number of records increases, the cost increases.

Table 7.1: The cost growth of memory usage and the min-max-query execution time of Stack 10 with respect to Stack 0 as the number of records in the base layer increases.

# of	Mem-Cost Growth (MB)			Time-Cost Growth (ms)		
Recs	DR	SB	DR w. $\bowtie$	DR	SB	DR w. $\bowtie$
1k	0.06	0.05	0.07	1	1	1
10k	0.4	0.4	0.4	1	1	1
100k	4	4	4	1	7	2
1m	38	38	42	1	103	13
10m	381	381	420	24	1,220	222
30m	1,163	1,163	1,280	232	3,795	728

7.2.3 Discussion

Using block referencing and DLIs to store intermediate results provide us with significant space-cost savings compared to materialization. We went from an estimated 90.3GB (using materialization) to 5.5GB (using block referencing and DLIs) to store in memory the original data set (4.3GB) and the results of 30 operators, each with an average of 20m records. The use of DLIs reduced the dereferencing-cost growth from linear to zero in the case where only operators with DR implementations are used. The dereferencing cost starts to grow slowly as we use more operators with SB implementations.

There is far more diversity in typical use-cases than the use-case we used in this test. For example, the number of records usually drops significantly as we add more levels to a given stack as a result of filtration and aggregation, which causes the space- and time-cost growth to drop significantly. We also see that data-analysis sessions usually involve a combination of vertical and horizontal expansions in the SQL Graph, digging deeper investigating one path of analysis versus trying alternative data analysis paths. As a result, stacks with large sizes (hundreds of layers) are rare (based on observation and experience), and even if we use many operators with SB

implementations, the dereferencing cost would still be below the interactive-speed threshold. The point is that the use-case we used is designed to be a worst-case scenario for a client-based data analysis. That is, having a stack of height 20 where the total number of rows at each level stays the same at around 30m records is rare. The reason it is rare is that part of analyzing data is producing results that are readable or comprehensible by a human, which involves reducing the number of rows as the analysis continues. In this worst-case-scenario use-case, we were able to achieve our goal of maintaining interactive speed while staying below space limit.

7.3 NAÏVE VERSUS CUSTOMIZED STORAGE ENGINE

In Chapter 6, we discussed two in-memory storage engines that we used to store the working data sets and the cached or materialized data. We talked about how inefficient the naïve approach is in terms of space using Java Objects. We also discussed another approach that uses low-level byte arrays to store the data in a compact way and eliminate the unnecessary overhead that is associated with using Java Objects. In this section we run an experiment to compare the two engines in terms of space and time.

7.3.1 Experiment Setup

The environment setup is as discussed in Section 7.1. We used the same highway data set that we used in the first experiment (Section 7.2). As mentioned before, the highway data set had an in-memory footprint of about 4.3GB using the naïve storage engine. We will see later the in-memory footprint of the same data set using the new, customized storage engine.

There are three categories of storage, 1) storing working data sets, 2) storing materialized data such as in `aggregate`, and 3) storing data-block references and DLIs. Neither the new, customized storage engine nor the old, naïve storage engine is involved in how data-block references or DLIs are stored. So Category 3 should

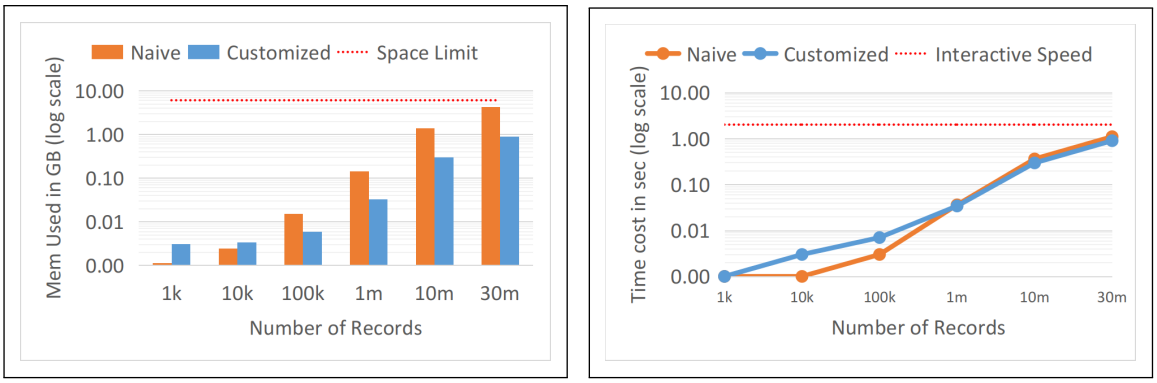
not be affected by the change in the underlying storage engine. However, Categories 1 and 2 use the exact same storage engine to store the data. So we only need to test one of the categories. Thus in this experiment, we perform the tests on only the working data set, or specifically, only the base layers.

The goal of the experiment is to compare the space and access-time cost of both storage engines. We first start with the full original data set of $\sim$ 30m records and run the min-max query that we discussed in the first experiment. We measure the space cost of storing the data set and the time it takes to complete the min-max query. We perform the same experiment again on the same data set but a reduced size. The sizes that we test are 1k, 10k, 100k, 1m, and 10m records.

7.3.2 Results

Table 7.2 lists the results of the space and access-time costs for the six (including the original data set) data-set sizes. Figure 7-4a shows the space cost comparison between the naïve (old) and the customized (new) storage engines. Figure 7-4b shows the access-time cost comparison. As you can see from the table, for large data sets, we managed to reduce the space cost by almost 80% with the new customized storage engine. Because there is a fixed storage cost for bookkeeping, the naïve storage engine starts to gain the upper hand over the customized one for small data sets ($<$ 4MB). However, the fixed storage cost is around 3MB. Data sets with such a small size are not a subject of concern unless we use hundreds or thousands of those data sets at once during the analysis, which we do not believe to be a realistic use-case.

In addition to space saving, the customized storage engine was slightly faster when accessing data for large data sets, and slightly slower for small data sets. Since access time on both engines is fast for small data sets, we do not see any advantage for using the naïve storage engine over the customized one. As you will see in the next section, the customized storage engine is more efficient in terms of space and time than Spark [71], PostgreSQL [28], and MySQL's in-memory tables [18]. (See Table



(a) Space cost (b) Access-time cost

Figure 7-4: A comparison between the naïve storage engine versus customized storage engine in terms of space cost and data-access-time cost. The comparison is using the same data set but with different sizes (varying the number of rows).

7.4 row #1 for space cost, and Table 7.5 min_max_query0 for access time.)

Table 7.2: The results of comparing the naïve storage engine and the customized storage engine in terms of space cost and data-access-time cost over data sets with different sizes (varying the number of rows).

# Recs	Space Cost (MB)		Access-Time Cost (ms)	
	Naïve	Customized	Naïve	Customized
1k	1.1	3.1	1	1
10k	2.4	3.3	1	3
100k	15.1	6.0	3	7
1m	142.0	32.5	36	34
10m	1,408.1	296.6	359	294
30m	4,291.3	896.6	1,094	894

7.3.3 Discussion

By reducing the size of the working data set (and the materialized data) to about 80% using the new storage, we provide significantly more space for data analysis than we would using the old storage. In addition, the new storage engine provides stable and predictable behavior in terms of space consumption regardless of the size of the data. On the other hand, using Java Objects for the old storage is unpredictable and can cause space-consumption explosions that consume the entire memory. These

explosions prevented us from testing realistic use-cases with large data sets. By using the new storage engine we were able to test realistic use-cases with big data, as we will see in the next section.

7.4 REALISTIC USE-CASE

The final experiment is to test a realistic use-case to give a sense of how the techniques we discussed in this research would work in real life. The goal for this experiment is to see how jSQL$_e$ compares to other known data-analysis systems in a real data-analysis use-case. Unlike the previous experiments, this experiment focuses more on build time than access time (in addition to space cost) because build time is what we can use to fairly compare jSQL$_e$ to other systems. We want to test the space cost and the build time of each system given that we want to keep the results of every single operator during the data-analysis process. Note that although we can measure access time in jSQL$_e$, we cannot measure pure access time in the other systems that we tested, at least not without hacking the code. The only way to access the data in the other systems is by executing a query, which involves going through the query optimizer, accessing the data, and processing the data to generate the results (build time). So we measured access time in this experiment using the build time of running min-max queries. (We will talk more about these queries in the Experiment Setup.) Note that for this experiment we actually build layers in jSQL$_e$, unlike the previous experiments.

At this point of the research, our prototype system, jSQL$_e$, had developed to a fully fledged data-manipulation system. We spent about three months optimizing the core algorithms of all the operators and adding features and utility functions that are typical in any data-manipulation system. The jSQL language also evolved to a rich and more complex language to support complex data-analysis needs. We also retired the old storage engine and we started using the new customized storage engine. Note that the development stages that jSQL$_e$ went through are typical and reasonable. It

would not have been worth investing in time and space optimizations if the basic operators did not work as we hypothesized.

7.4.1 Experiment Setup

In this experiment we use a real use-case that we describe in detail in the Appendix. The short story is as follows. This data-analysis use-case was part of a past class project. The goal was to create a model that uses historical transit data to predict future (up to a week from the present) arrival times for buses and trains. For a given route (bus or train), a stop, and a schedule time, our goal is that the model predicts the arrival time for the bus or the train within ± 3 minutes from the actual arrival time. We used about six months worth of data from TriMet [64] (Portland, Oregon's public transit system). The data that is being captured is individual bus and train arrivals and departures at a given stop at a given schedule time. The data schema is described in Appendix A.2. Originally, the analysis was done using PostgreSQL [28]. We used PostgreSQL as it is intended as a relational database management system. That is, we have the data in tables, and then we issue complex queries to get results. As the analysis progresses, we modify the queries and run them again to explore various options. The original analysis (every query that we issued during the entire data-analysis exploration session) that we did using PostgreSQL is listed in Appendix A.3.

The idea for this experiment is to take the same analysis and try to replicate it using jSQL$_e$. However, this time we will use jSQL$_e$ as it is intended. That is, keep all intermediate results in main memory and try to reuse as many of these results as possible. The process requires us to break down the original queries into individual operators, store the results of each operator, and reuse the stored results whenever possible. As a comparison, we will try to simulate, as much as possible, what jSQL$_e$ does in three other systems: PostgreSQL, Spark [71], and MySQL [18] and measure their performance against jSQL$_e$ in terms of space and build time.

The three systems that we chose each serve a unique purpose in our comparison. PostgreSQL is the system that we used for the original analysis and we also used it for the simulated analysis. Although PostgreSQL does not support in-memory tables, we wanted to use it as a baseline in terms of the amount of space that it takes to store each intermediate result and in terms of the time it takes to build each result given that the input data is cached or materialized at the input layer. That is, in terms of space, we can see the actual space cost of each result, and in terms of time, we can see the pure build time without any extra overhead, such as dereferencing costs in jSQL$_e$, and with decades of optimizations to the core algorithms of each operator. Also since PostgreSQL is a disk-based system, it can give us a sense of what the effect of using disk would be if jSQL$_e$ were to use disk for storage as a fallback mechanism. MySQL is a step above PostgreSQL in that it provides similar capabilities to PostgreSQL but it offers in-memory tables. Spark is a system that is the closest we can find to a system like jSQL$_e$.

Spark supports out-of-the-box caching of intermediate results and keeps track of the lineage of each operator. Moreover, Spark is designed to be used with in-memory storage (with the option to use disk as well). However, Spark, as far as we can tell, does not try to reduce the space cost of these intermediate results (aside from giving the option to compress the data), if the user chooses to keep them around. If an intermediate result is needed at a later step and it is cached in memory, Spark uses that data; otherwise, it uses the result's lineage to recompute the result's data. This behavior is the closest we can get to a system that is similar to jSQL$_e$ but without the space-saving techniques that are the core of this research.

In all three systems (in addition to jSQL$_e$), we did not add any explicit optimizations in terms of space or time. For example, we did not create indexes, cluster data, or partition the tables to speed up data access. All systems use their default settings and whatever optimizations the query optimizers can figure out on their own to best process the data. Moreover, each system ran in a single-worker environment, or to

be more specific, only one CPU core was utilized at any given time. For Spark, we used two configurations, one that used only memory to store the data, and the other used a hybrid of memory and disk. The hybrid option allows Spark to use memory, and when it runs out of memory, it uses disk as a temporary buffer to store unused data (data that is not needed for the operation at hand). With jSQL$_e$, Spark, and MySQL, we set the memory limit to 6GB.

The data set that we used had a size of 1.3GB in a CSV file format. When we talk about results, we will see the space cost of this data set once it was loaded into each system. The data set consisted of about 33 million records, and its schema is described in Appendix A.2. The original data-analysis, as listed in Appendix A.3, consists of a total of 27 complex queries. We refer to each of these queries as a *statement* (STMT); so "`STMT 1`" refers to the first query in the original analysis, "`STMT 2`" refers to the second query, and so on. To fit the jSQL$_e$ data model, we broke down each of these statements into individual operators so that each operator became a query of its own, as listed in Appendix A.4. The result of each operator, a data layer, was given a name. For example in the query "`route58_stop910 = SELECT stop_events WHERE ...`", `route58_stop910` is the result's name (the data-layer id). We refer to each one of these results as a *step* in the data-analysis process. Each statement now represents a stack of data layers. There is a total of 178 steps that makeup the original 27 statements (27 stacks).

Table 7.3 summarizes the data analysis process. The analysis process is described in detail in Section A.3. The first column (STMT) is the statement number. The second column (#) is the step number. The third column (Data Layer Id) is the name of the intermediate result. The forth column (Type) is the type of the operator that was applied at that step. The last column (#Rows) is the number of rows that resulted from that operator. The first row (`stop_events`) is the original data set once it was loaded into the system. Figure 7-5 shows the full SQL Graph for all 178 layers. As you can see from the graph, the analysis provides vertical as well as horizontal

expansion. You can also see that in many steps (e.g., #36), the analysis branches out from previous paths to explore other paths.

For the other systems, we took each one of these jSQL$_e$ steps and wrote the equivalent query in the corresponding system and forced the system to store or cache the results. For PostgreSQL, we store the result of each query (jSQL-equivalent query to each of the 178 layers) in a table (the data is stored on disk) that has the same name as the corresponding data-layer id, as listed in Appendix A.6. For MySQL, we did the same thing we did with PostgreSQL, but we used in-memory tables instead (the data is stored in a table that resides only in memory), as listed in Appendix A.5. For Spark, we used `DataFrames`, which allowed us to write regular SQL queries, as listed in Appendix A.7. Each result in Spark is referred to as a *view*; the name of each view is the corresponding data-layer id. Spark supports caching of intermediate results out-of-the-box, and it supports multiple storage levels. We tested two of those storage levels, one where we told Spark to cache the result of each view only in memory, and the other where we told Spark to utilize both memory and disk to cache the results.

In PostgreSQL, MySQL, and Spark, you will see that there are steps that we skipped (e.g., Step #6). There are two situations that result in steps being skipped. The first is where we have a jSQL$_e$ operator for which we do not have an equivalent in other systems, such as `group`. In all three systems, there is the `group by` operator, which is equivalent (almost[4]) to a `group` followed by an `aggregate` operator in jSQL$_e$. The other situation is when an operator is not necessary. For example, the `aggregate` operator in jSQL$_e$ keeps the group column, which `group by` does not do. If we want the result of an `aggregate` to be 100% equivalent to the result of a `group by`, we have to project away the group column right after the `aggregate` operator, such as in

[4]In jSQL$_e$, the `group` operator, in addition to the grouping columns, creates the group column. The `aggregate` operator, if a group column is given, produces a schema that is equivalent to the input schema plus a column for each of the aggregation functions. So if a `group` followed immediately by an `aggregate`, the result is equivalent to SQL's `group by` operator's result plus the group column from jSQL$_e$'s `group` operator.

the steps sequence #74 (`group`), #75 (`aggregate`), and #76 (`project` to exclude the group column).

For all four systems, the goal was to measure three things:

1. The space cost of storing each intermediate result in addition to the original data set.

2. The build time for each intermediate result.

3. The build time for running a min-max query at the top of each of the 27 stacks, in addition to the original data set that we refer to as Stack 0. Similar to the previous experiments, we use the min-max query to measure access time. However, as we mentioned earlier, we cannot measure pure access time for the other systems. So we used build time—the time it takes to construct the results—for all four systems. Note that for jSQL$_e$, build time is the time it takes to run the `aggregate` operator and build the `aggregate` layer, whereas for the other three systems, build time is the time it takes to construct the results for the SQL query (no tables are created).

For all systems except PostgreSQL, the goal is to try to force the system to keep all intermediate results in memory. However, as you will see in a bit, each of the four systems interprets and handles such a requirement differently, which affects the overall build time. Next we present and discuss the results of the experiment.

Table 7.3: A list of the data layers (or equivalent tables in other systems) that were generated during the data analysis process. For more information on each layer, see Appendix A.4

STMT	#	Data Layer Id	Type	#Rows
-	1	stop_events	IMPORT	32,950,296
1	2	route58_stop910	SELECT	36
	3	route58_stop910_ordered	ORDER	36
2	4	stop9821	SELECT	117
	5	distinct_routes_at_stop9821	DISTINCT	2
3	6	unique_stops	GROUP	27,572,655
	7	unique_stops_count	AGGREGATE	27,572,655
	8	duplicates	SELECT	3,873,624
4	9	route58_loc12790	SELECT	2
5	10	stop9818	SELECT	65
	11	distinct_routes_at_stop9818	DISTINCT	1
6	12	stop_events_with_dow	PROJECT	32,950,296
	13	stop_events_with_dow_group	GROUP	11,704,508
	14	stop_events_with_dow_histogram	AGGREGATE	11,704,508
7	15	model1_v1_avg_delay_per_dow	SELECT	13,260,965
	16	model1_v1_avg_delay_per_dow_group	GROUP	2,513,507
	17	model1_v1_agg	AGGREGATE	2,513,507
	18	model1_v1_proj	PROJECT	2,513,507
	19	model1_v1	PROJECT	2,513,507
8	20	model1_v2_select_base_data	SELECT	13,065,538

STMT	#	Data Layer Id	Type	#Rows
	21	model1_v2_select_base_data_with_delay	PROJECT	13,065,538
	22	model1_v2_select_base_data_group	GROUP	11,143,819
	23	model1_v2_cleaned_base_data	AGGREGATE_REF	12,851,000
	24	model1_v2_base_model_group	GROUP	2,513,467
	25	model1_v2_base_model	AGGREGATE	2,513,467
	26	model1_v2_final_res_join	JOIN_LEFT	10,561,687
	27	model1_v2_final_res_group	GROUP	2,513,467
	28	model1_v2_final_res_agg	AGGREGATE	2,513,467
	29	model1_v2	PROJECT	2,513,467
9	30	model1_v2_compare_sel_route	SELECT	31,706
	31	model1_v2_compare_sel_dow_tue	SELECT	4,567
	32	model1_v2_compare_sel_dow_wed	SELECT	4,568
	33	model1_v2_compare_join	JOIN_INNER	4,566
	34	model1_v2_compare_project	PROJECT	4,566
	35	model1_v2_compare	ORDER	4,566
10	36	model2_v2_select_base_data_group	GROUP	11,143,819
	37	model2_v2_cleaned_base_data	AGGREGATE_REF	12,851,000
	38	model2_v2_base_model_group	GROUP	937,079
	39	model2_v2_base_model	AGGREGATE	937,079
	40	model2_v2_final_res_join	JOIN_LEFT	10,823,144
	41	model2_v2_final_res_group	GROUP	937,079
	42	model2_v2_final_res_agg	AGGREGATE	937,079

STMT	#	Data Layer Id	Type	#Rows
	43	model2_v2	PROJECT	937,079
11	44	model2_v2_2_avg_delay_per_dow_class	SELECT	22,633,386
	45	model2_v2_2_avg_delay_per_dow_group	GROUP	1,098,569
	46	model2_v2_2_agg	AGGREGATE	1,098,569
	47	model2_v2_2	PROJECT	1,098,569
	48	model2_v2_2_proj	PROJECT	1,098,569
12	49	compare_v2_m1_m2_sel_m1	SELECT	65
	50	compare_v2_m1_m2_sel_m2	SELECT	461,575
	51	compare_v2_m1_m2_join	JOIN_INNER	65
	52	compare_v2_m1_m2_project	PROJECT	65
	53	compare_v2_m1_m2	ORDER	65
13	54	baseline_l1	SELECT	10,316,910
	55	baseline_l2	GROUP	2,122
	56	baseline_l3	AGGREGATE	2,122
	57	baseline_l4	PROJECT	2,122
	58	baseline_l5	GROUP	2
	59	baseline_l6	AGGREGATE	2
	60	baseline_l7	PROJECT	2
	61	baseline_l8	ORDER	2
14	62	baseline_rush_hour_l1	SELECT	3,173,612
	63	baseline_rush_hour_l2	GROUP	611
	64	baseline_rush_hour_l3	AGGREGATE	611
	65	baseline_rush_hour_l4	PROJECT	611

STMT	#	Data Layer Id	Type	#Rows
	66	baseline_rush_hour_l5	ORDER	611
15	67	predicting_feb_arrival_l1	SELECT	9,968,658
	68	predicting_feb_arrival_l2	PROJECT	9,968,658
	69	predicting_feb_arrival_l3	JOIN_INNER	9,944,193
	70	predicting_feb_arrival_l4	PROJECT	9,944,193
	71	predicting_feb_arrival_l5	GROUP	1,659
	72	predicting_feb_arrival_l6	AGGREGATE	1,659
	73	predicting_feb_arrival_l7	PROJECT	1,659
	74	predicting_feb_arrival_l8	GROUP	8
	75	predicting_feb_arrival_l9	AGGREGATE	8
	76	predicting_feb_arrival_l10	PROJECT	8
	77	predicting_feb_arrival_l11	ORDER	8
16	78	predicting_feb_arrival_rush_hr_l1	SELECT	3,173,612
	79	predicting_feb_arrival_rush_hr_l2	PROJECT	3,173,612
	80	predicting_feb_arrival_rush_hr_l3	JOIN_INNER	3,165,922
	81	predicting_feb_arrival_rush_hr_l4	PROJECT	3,165,922
	82	predicting_feb_arrival_rush_hr_l5	GROUP	613
	83	predicting_feb_arrival_rush_hr_l6	AGGREGATE	613
	84	predicting_feb_arrival_rush_hr_l7	PROJECT	613
	85	predicting_feb_arrival_rush_hr_l8	ORDER	613
17	86	predicting_feb_arrival_dow_class_l1	JOIN_INNER	9,964,598
	87	predicting_feb_arrival_dow_class_l2	PROJECT	9,964,598
	88	predicting_feb_arrival_dow_class_l3	GROUP	1,299

STMT	#	Data Layer Id	Type	#Rows
	89	predicting_feb_arrival_dow_class_l4	AGGREGATE	1,299
	90	predicting_feb_arrival_dow_class_l5	PROJECT	1,299
	91	predicting_feb_arrival_dow_class_l6	GROUP	8
	92	predicting_feb_arrival_dow_class_l7	AGGREGATE	8
	93	predicting_feb_arrival_dow_class_l8	PROJECT	8
	94	predicting_feb_arrival_dow_class_l9	ORDER	8
18	95	predicting_feb_arrival_rush_hr_dow_class_l1	JOIN_INNER	3,172,691
	96	predicting_feb_arrival_rush_hr_dow_class_l2	PROJECT	3,172,691
	97	predicting_feb_arrival_rush_hr_dow_class_l3	GROUP	577
	98	predicting_feb_arrival_rush_hr_dow_class_l4	AGGREGATE	577
	99	predicting_feb_arrival_rush_hr_dow_class_l5	PROJECT	577
	100	predicting_feb_arrival_rush_hr_dow_class_l6	ORDER	577
19	101	model1_v3_l1	GROUP	11,143,819
	102	model1_v3_l2	AGGREGATE	11,143,819
	103	model1_v3_l3	PROJECT	11,143,819
	104	model1_v3_l4	GROUP	2,513,467
	105	model1_v3_l5	AGGREGATE	2,513,467
	106	model1_v3_l6	JOIN_INNER	11,143,819
	107	model1_v3_l7	GROUP	2,513,467
	108	model1_v3_l8	AGGREGATE	2,513,467
	109	model1_v3_l9	PROJECT	2,513,467
20	110	model2_v3_l1	PROJECT	11,143,819
	111	model2_v3_l2	GROUP	937,079

STMT	#	Data Layer Id	Type	#Rows
	112	model2_v3_l3	AGGREGATE	937,079
	113	model2_v3_l4	JOIN_INNER	11,143,819
	114	model2_v3_l5	GROUP	937,079
	115	model2_v3_l6	AGGREGATE	937,079
	116	model2_v3_l7	PROJECT	937,079
21	117	baseline_v2_l1	SELECT	10,173,881
	118	baseline_v2_l2	GROUP	8,649,975
	119	baseline_v2_l3	AGGREGATE	8,649,975
	120	baseline_v2_l4	PROJECT	8,649,975
	121	baseline_v2_l5	PROJECT	8,649,975
	122	baseline_v2_l6	GROUP	431
	123	baseline_v2_l7	AGGREGATE	431
	124	baseline_v2_l8	PROJECT	431
	125	baseline_v2_l9	ORDER	431
22	126	baseline_v2_rush_hour_l1	SELECT	3,173,612
	127	baseline_v2_rush_hour_l2	GROUP	2,737,331
	128	baseline_v2_rush_hour_l3	AGGREGATE	2,737,331
	129	baseline_v2_rush_hour_l4	PROJECT	2,737,331
	130	baseline_v2_rush_hour_l5	PROJECT	2,737,331
	131	baseline_v2_rush_hour_l6	GROUP	201
	132	baseline_v2_rush_hour_l7	AGGREGATE	201
	133	baseline_v2_rush_hour_l8	PROJECT	201

STMT	#	Data Layer Id	Type	#Rows
	134	baseline_v2_rush_hour_l9	ORDER	201
23	135	comp_predic_v2_l1	PROJECT	8,649,975
	136	comp_predic_v2_l2	JOIN_INNER	8,629,875
	137	comp_predic_v2_l3	PROJECT	8,629,875
	138	comp_predic_v2_l4	GROUP	1,043
	139	comp_predic_v2_l5	AGGREGATE	1,043
	140	comp_predic_v2_l6	PROJECT	1,043
	141	comp_predic_v2_l7	GROUP	479
	142	comp_predic_v2_l8	AGGREGATE	479
	143	comp_predic_v2_l9	PROJECT	479
	144	comp_predic_v2_l10	ORDER	479
24	145	comp_predic_v2_rush_hour_l1	PROJECT	2,737,331
	146	comp_predic_v2_rush_hour_l2	JOIN_INNER	2,731,048
	147	comp_predic_v2_rush_hour_l3	PROJECT	2,731,048
	148	comp_predic_v2_rush_hour_l4	GROUP	543
	149	comp_predic_v2_rush_hour_l5	AGGREGATE	543
	150	comp_predic_v2_rush_hour_l6	PROJECT	543
	151	comp_predic_v2_rush_hour_l7	GROUP	279
	152	comp_predic_v2_rush_hour_l8	AGGREGATE	279
	153	comp_predic_v2_rush_hour_l9	PROJECT	279
	154	comp_predic_v2_rush_hour_l10	ORDER	279
25	155	comp_predic_v3_l1	PROJECT	8,649,975
	156	comp_predic_v3_l2	JOIN_INNER	8,643,458

STMT	#	Data Layer Id	Type	#Rows
	157	comp_predic_v3_l3	PROJECT	8,643,458
	158	comp_predic_v3_l4	GROUP	1,034
	159	comp_predic_v3_l5	AGGREGATE	1,034
	160	comp_predic_v3_l6	PROJECT	1,034
	161	comp_predic_v3_l7	GROUP	484
	162	comp_predic_v3_l8	AGGREGATE	484
	163	comp_predic_v3_l9	PROJECT	484
	164	comp_predic_v3_l10	ORDER	484
26	165	comp_predic_v3_rush_hour_l1	PROJECT	2,737,331
	166	comp_predic_v3_rush_hour_l2	JOIN_INNER	2,735,673
	167	comp_predic_v3_rush_hour_l3	PROJECT	2,735,673
	168	comp_predic_v3_rush_hour_l4	GROUP	497
	169	comp_predic_v3_rush_hour_l5	AGGREGATE	497
	170	comp_predic_v3_rush_hour_l6	PROJECT	497
	171	comp_predic_v3_rush_hour_l7	GROUP	258
	172	comp_predic_v3_rush_hour_l8	AGGREGATE	258
	173	comp_predic_v3_rush_hour_l9	PROJECT	258
	174	comp_predic_v3_rush_hour_l10	ORDER	258
27	175	comp_pred_model1_and_model2_l1	JOIN_INNER	3,286,485
	176	comp_pred_model1_and_model2_l2	SELECT	65
	177	comp_pred_model1_and_model2_l3	PROJECT	65
	178	comp_pred_model1_and_model2_l4	ORDER	65

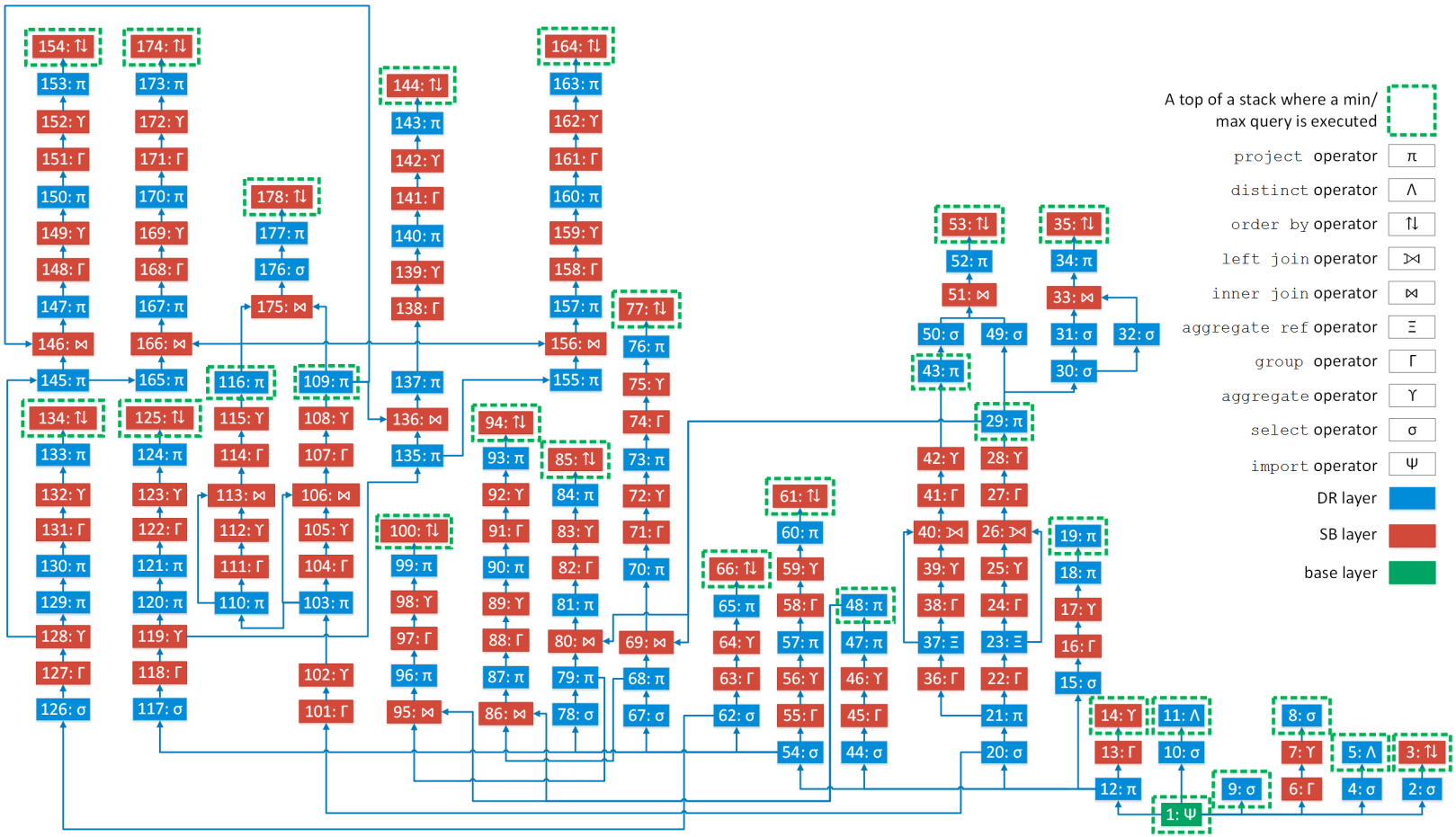


Figure 7-5: The SQL Graph of the realistic use-case discussed in the Appendix.

7.4.2 Results

The space cost and the build time results for each of the 178 steps are listed in Table 7.4. The first column (#) is the step number from Table 7.3 (column #). Note that we did not list the build time for Spark. We will talk about why in a bit when we discuss the results and behavior of each system. But for now, the reason has to do with Spark's lazy evaluation, which prevented us from measuring the build time for the individual steps, at least not in a way that would make the measurements a fair comparison. Figure 7-6 shows the cumulative space cost for all four systems as the data-analysis progresses with each step. The secondary y-axis to the right is for the number of rows (the bars) that each step generated. Figure 7-7 shows the cumulative build time for all four systems as the data-analysis progresses with each step. The secondary y-axis to the right is also for the number of rows (the bars) that each step generated.

The first row in Table 7.4 is the cost of loading the original data set into the system. We do not have build time for the first row because there was no data processing involved; it was only loading the data into its appropriate storage inside each system. The data-loading time was more or less the same for all three systems (jSQL$_e$, PostgreSQL, and MySQL; Spark had a mind of its own, we will see why in a bit). The space cost for Step #1 shows the efficiency of the storage engine in each system. As you can see, jSQL$_e$'s customized storage engine is the most efficient, though Spark and MySQL are very close. Keep in mind that none of the systems uses compression. PostgreSQL, on the other hand, takes almost twice as much storage. The extra cost is understandable since PostgreSQL is a read-write disk-based storage and, therefore, it has a lot more bookkeeping to do than the other systems.

Table 7.5 shows the results of running the min-max queries that are listed in Appendix A.4.1, A.5.1, A.6.1, and A.7.1. The first column (Query) is the stack (STMT) on which the query was run. The second column (Input Layer #) is the step number from Table 7.3, which indicates the intermediate result (the top of the

stack) on which the query was issued. The third column (#Rows) shows the number of rows that the query has to access. Since access time in jSQL$_e$ depends on the stack height and how many DR and SB layers are in it, the columns (strictly for jSQL$_e$) Stack Height, #DR Layers, and #SB Layers show statistics about the stack on which the query was issued; Figure 7-8 provides a visual representation for these statistics and the cumulative build time for jSQL$_e$. Finally, the remaining columns show the time it took to run the min-max query in each system (Spark with both configurations, memory only and hybrid); Figure 7-9 provides a visual representation for the cumulative min-max-query build time in all four systems (Spark with both configurations).

The first query `min_max_query0` is for the original data set to get a sense of the pure build-time cost without the extra overhead from dereferencing data blocks. Note that every other system besides jSQL$_e$ did not have the extra dereferencing overhead since the data is materialized at the input table. Also note that every system other than Spark had its input data (top layer in the stack) to the min-max query already built. Spark, as we will discuss in a bit, had to wait until we issue the min-max query to generate many or all the results in the stack, hence the high build-time cost. We assume that the user builds the results (the layers) one at a time, as it is the case with exploratory data analysis. In that case, Table 7.5 gives us a sense of what the user would experience using any of these systems to access the data. However, if the user submits all the queries to the systems at once, Tables 7.4 and 7.8 and Figure 7-7 would be more accurate in representing the user experience. Next we discuss the results and the behavior of each system.

jSQL: In terms of space cost, as expected, jSQL$_e$ came in on top at almost every step, and by large margins. The total space cost for all 178 data layer (including the original data set) is about 4.6GB, as listed in Table 7.7. The cost includes the steps that are skipped in other systems, such as `group` operators. As you can see from Table

7.6, the overall space cost for all the `group` operators is about 1.6GB, which makes up about 36% of the total space cost. So the more comparable space cost to the other systems would be about 3GB instead of 4.6GB. Although we added the `group` operator in jSQL$_e$ data model so that we can reuse the results of the grouping more than once, in this particular data-analysis use-case, the results of the `group` operators were strictly used as inputs to the immediately following `aggregate` or `aggregate ref` operators. So keeping the `group` operators' results in this use-case was a waste of space. This observation opens the door to strategies that we can use to better utilize the space.

As we mentioned earlier, build time consists of data-access time (the main focus of this research) and running the core algorithm of the operator. We only spent three months on optimizing the core algorithms of each operator to bring down the build time to practical numbers that are comparable to other systems. There is still a lot of room for improvement, and the numbers that you see in Table 7.4 can be brought down much further. Even with that short period spent on optimization (compared to decades for other systems; Spark has been around for only a decade), jSQL$_e$ still put up a good fight even with the dereferencing cost that was added to data-access time. However, looking at Figure 7-7, you can see that the cumulative build time starts off close to the other systems, but it starts to diverge at Step #69, which is when we used a `join` operator with an input that has a stack height of 12 layer, 7 of which are SB layers, as you can see from Figure 7-5.

The jump implies that dereferencing cost played a role, but it is not clear what the percentage is. What we do know, however, is that when we used a nested-loop join for the core algorithm of the `join` operator, the build time was about 9 hours (not a big surprise given that it is a $O(N^2)$ algorithm). After adding a query optimizer to recognize the cases where we can use a sort-merge join instead, the build time went down to a little over 2 minutes. Clearly there is more that can be done to improve the `join`'s core algorithm. Looking at the overall build time for jSQL$_e$,

as listed in Table 7.8, it took almost twice as long as PostgreSQL to run the entire data-analysis process, but was a bit faster than Spark with memory-only configuration. Although PostgreSQL finished the analysis in half the time, PostgreSQL was only able to achieve that because we materialized all the intermediate results (there was no dereferencing cost associated with accessing the data). That materialization cost almost 10 times the space cost that jSQL$_e$ required, as you can see in Table 7.7. In other words, jSQL$_e$ spent twice the time cost, but saved 90% of the space cost. For Spark, which is more comparable to jSQL$_e$, jSQL$_e$ did more or less the same in terms of time cost, but saved about 73% of the space cost (Spark required almost 4 times as much space, or almost 6 times if we ignore the `group` operators).

PostgreSQL: In terms of space cost, PostgreSQL was the worst, as you can see from Table 7.4, for individual steps, and from Table 7.7, for the overall space cost. However, for data sets that are less than 1MB, in many steps, PostgreSQL seems to be doing better than Spark and MySQL. It is not clear why, but it could be because Spark and MySQL have some fixed cost that is associated with a certain operator regardless of the size of the data itself. It is no surprise that PostgreSQL took the most amount of space, since it provides read-write, disk-based storage. Although it is not fair to compare jSQL$_e$ to PostgreSQL, PostgreSQL's space cost provides us with a standard space cost that we need to store the materialized intermediate results. We can use this standard space cost to measure the efficiency of the space-saving techniques that jSQL$_e$ is using. As we mentioned before, these techniques that we used in jSQL$_e$ saved us 90% of the standard space cost, see Table 7.7 for a comparison between the systems.

In terms of build time, we did not expect PostgreSQL to do as well as it did. In fact PostgreSQL, overall, was the fastest in terms of build time; though MySQL was faster for the duration that it lasted. The reason we did not expect PostgreSQL to do as well as it did is that it is a disk-based storage. However, once we looked deeper,

it was clear why PostgreSQL did well. There are two main factors that contributed to such performance. The first is the common, known technique that all disk-based data management systems use, which is data buffering to overcome disk inefficiency. The short story is that data management systems maintain a fixed space in memory (a buffer) and load data into this buffer usually multiple pages (a page is a set of records) at a time. If the buffer becomes full, a cold page (a page that has not been accessed recently) is evicted (sent back to disk) and another page takes its place. So if the data that is needed for the current operation is warm (already in the buffer), the system performs more or less as an in-memory system, which brings us to the second factor.

If we look at Figure 7-5, we can see that the input data in about 85% of the operations is the result of the previous operation. That is, the required data for 85% of the operations is almost certainly available in the buffer. The other 15% of the operations depend on how long their input data had been sitting in the buffer and the size of the results that came after it. The behavior that the results of 85% of the operators are the inputs to the next operator is not unique to this use-case. Data analysis is not random in nature, it is exploratory. That is, the analyst picks a path and continues to explore that path until something happens that causes the starting of another path or branching from an existing path. So the vast majority of the data-analysis process is spent on extending paths (operating on the previously acquired results) instead of exploring alternative ones. This observation is important because it means that using disks to aid data storage in exploratory data-analysis systems, such as jSQL$_e$, is more or less as efficient as using pure in-memory storage.

Overall, PostgreSQL was about twice as fast as jSQL$_e$ in terms of build time. However, the input data to the operators in PostgreSQL were materialized, which reduced access-time cost. On the other hand, jSQL$_e$ had the extra dereferencing cost, which grew as the analysis progressed. The biggest difference in build-time cost was with the joins and the groupings. In addition to materialization, PostgreSQL has

had decades to develop and optimize its operators' algorithms. Although we can take advantage of some of PostgreSQL's optimizations and transfer that to jSQL$_e$, there are many optimization techniques that simply will not work as is; the two systems are built for different infrastructures. PostgreSQL is built to store data on disk and allow read-write operations. On the other hand, jSQL$_e$ is built for in-memory storage and read-only operations. Moreover, jSQL$_e$ uses block referencing and an imperative query language, as opposed to materialization and a declarative query language in PostgreSQL.

For the joins, jSQL$_e$ has twice the dereferencing cost since there are two inputs. However, we believe that with more time on optimizing the join algorithms, we can reduce this cost considerably. For grouping operations, remember that jSQL$_e$ has a `group` operator and an `aggregate` operator, whereas PostgreSQL (and the others) has a `group by` operator. The majority of the cost in a `group by` operator is spent on the group part of the operator. Although in many steps, jSQL$_e$ is faster when doing the `aggregate` operator, the actual comparable cost to PostgreSQL's `group by` is the sum of both costs of jSQL$_e$'s `group` and `aggregate` operators. The reason why the grouping is faster in PostgreSQL is that PostgreSQL uses hashing to create the groups, while jSQL$_e$ uses a sort-merge-based algorithm because we found it to be far more space efficient and far more predictable than hashing. Although we believe that we can make the algorithm faster in jSQL$_e$, theoretically, it will not be faster or as good as hashing, if given the proper amount of space. However, we believe that the predictability and the low space-cost of a sort-merge-based algorithm far outweighs the extra speed we get from hashing.

Spark: We were most interested to see how Spark behaves compared to jSQL$_e$. However, we soon came to realize that the two systems were too different in their behavior. Despite our efforts to try to simulate jSQL$_e$ behavior in Spark, we hit many brick walls that prevented us from providing a good comparison. The biggest

impediment we faced was lazy evaluation. In Spark, data operators are not executed until either a `show` or a `store` command is issued. That is, until you want to view the data or store it on some medium (e.g., export the data to disk), Spark will only create execution plans for queries that you issue. For example, if you issue query A then you use the result of A in query B, neither query A nor B is executed even if you ask Spark to persist the results. If you ask to see the results of B, both A and B are executed at the same time. This behavior meant that we could not measure build time properly for each of the steps that we used in the data analysis.

We could have stored the result of each step to a disk, but that would have added a significant overhead to the build time. We could have also issued a `show` command to the results of each step, but many of those steps had millions of records, and displaying them all would not be feasible. Limiting the number of records to display (e.g., the first 100 records only) makes Spark process just enough data to generate that number of records; so that was not an option either. The only solution that we came up with to force Spark to process all the data at every step and force it to cache the data of each step is by issuing the `show` command only on the min-max queries. Because the min-max queries are chosen so that every record at the top of the stack is accessed, it means that every query in that stack has to be fully processed. But it also means that we can measure the build time only for the entire stack (a set of steps) and not for the individual steps, hence why there is no build-time column for Spark in Table 7.4. However, in Table 7.5, the Spark build-time is the time it took to run the entire stack for a given min-max query.

As we mentioned earlier, we used two configurations for Spark, one that uses only memory and one that is a hybrid of memory and disk. Spark is designed to be an in-memory data-analysis system. So what happens if Spark runs out of memory? If we asked Spark to use only memory, Spark starts to throw away the oldest data that is not needed for the current operation. Because Spark keeps track of the lineage of each result using RDDs [71], if Spark needs that thrown-away result later, it will have

to recompute it. How far back Spark has to go to recompute the result depends on what data is currently available in memory. For example, if we have a stack of 10 steps and we want to use Step #10's results that are not in memory, Spark will find the closest step whose results are still in memory and recompute Step #10's results from there.

On the other hand, if we ask Spark to use memory and disk, instead of throwing away the results, Spark will store them on disk. If those results are needed later, it will load them back into memory. Surprisingly, the hybrid configuration seems to be, overall, a bit faster than the memory-only configuration, as shown in Table 7.8. Obviously this observation is not a general rule and might not always be the case. As we mentioned, the cost of recomputing a result depends on how far back Spark has to go to find data that is in memory. For some cases, recomputing the result can be faster than loading the data from disk, and for other cases, the opposite can be true. Even in our use-case, you can see from Table 7.5 that, for example, Spark_m (memory-only configuration) is faster than Spark_{m+d} (hybrid configuration) in `min_max_query15` but not in `min_max_query19`.

The other issue that we faced with Spark is how Spark uses memory. Although we configured Spark to use only 6GB of memory as a maximum limit, Spark uses about 40% of that space for its internal uses (loading JVM classes and space that is needed to operate other components of the system), which, in our use-case, left about 3.6GB for data storage. This significant initial cost of loading the system means far less space to use for data analysis, especially in a client-based environment, which Spark is not designed for. The limited space that was left for data sometimes made the memory-only configuration less efficient compared to the hybrid one because Spark is now more likely to recompute results.

In terms of space cost, Spark with memory-only configuration was able to keep only 3.6GB of data at a time. Spark with a hybrid configuration kept all the data, but the data is spread between memory and disk. In both cases, the space cost of the

results of each step is the same whether the data is in memory or on disk. Overall, Spark seems to require half the space that PostgreSQL requires, as you can see from Table 7.7. Although we could not finish the analysis in MySQL (we will talk about why in bit), for those steps that we managed to do in MySQL, overall, Spark was slightly better. However, Spark was nowhere near as efficient in terms of space as jSQL$_e$, which is to be expected since Spark does not employ any special techniques to reduce the cost, besides offering to compress the data if the user wants.

In terms of build time, as we mentioned, we could not measure build time for individual steps. Instead, we relied on the build time of each min-max query (Table 7.5) and the overall runtime to perform the entire data analysis (Table 7.8). Although the min-max-query build time does not provide a good step-by-step comparison, it provides an important observation about exploratory data analysis and the use of lazy evaluation. Lazy evaluation makes sense for a system that is designed to work in a server-based environment where the data-analysis plan is built in advance and then sent to the server to be executed. In a client-based exploratory data analysis, you figure out the plan as you go. Part of figuring out the next step in the process is to examine the results so far by various tools (e.g., visualizations). So it is counterproductive to wait until the user wants to see the data (in some form or another) to start processing data. On the other hand, if we start to process the data as the instructions come in, we can spread the data-processing cost over time so that by the time the user wants to see the results, the data can be available within interactive speed. The min-max queries demonstrate this observation. Since Spark waited until the last minute to compute everything in the stack, it took Spark, in many cases, minutes to produce results. On the other hand, the other systems produced results more or less within interactive speed, as shown in Table 7.5 and in Figure 7-8.

Although we believe that full lazy evaluation is not suitable for an exploratory data-analysis environment, semi-lazy evaluation could improve performance (space and time). It could be more efficient to wait until we construct 2 to 3 layers before we

start processing the data and generating their results. We talk about this approach more later in Chapter 10 when we discuss future work. But the point is, we might have closed the door on full lazy evaluation, we believe that the door is still open for semi-lazy evaluation.

MySQL: Before we started the analysis, we did not expect much from MySQL, and the results, more or less, matched our expectations. However, we expected MySQL to last a bit longer than it did. Since MySQL does not have a fallback plan for when memory becomes full, any subsequent attempt to store data in memory will fail. For MySQL, we forced each intermediate result to be stored in an in-memory table. After Step #25, the space cost exceeded 6GB and, therefore, MySQL could not continue to process the subsequent steps, hence why there are no results in 7.4 after Step #25. However, for the steps that MySQL managed to do, the space cost was comparable to that of Spark and the build time was comparable to that of PostgreSQL. So it seems that if we have a large enough memory, MySQL would have outperformed both Spark and PostgreSQL if we consider both space cost and build time. However, having a large memory usually is not an option (at least not yet) for a client-based environment. So what other options do we have?

If we use disks as a fallback plan, we either end up with a system like PostgreSQL or a system like Spark (with the hybrid configuration). If we want to stay with memory only, we could end up with a system like Spark (with memory-only configuration) where we throw away old results and recompute them if we need them later. Or we could employ many of the techniques that jSQL$_e$ has to reduce the footprint of each intermediate result. Whichever choice we pick, MySQL will not continue to perform the same as it is performing now to support in-memory storage.

Table 7.4: The build time and the space cost of each intermediate result (data layers in jSQL$_e$ and tables in other systems). The first column is the number of the intermediate results in in Table 7.3. For build time, Spark is not shown because Spark does lazy evaluation and only builds the results when we perform the min-max queries. For some intermediate results, such as `group` operators (e.g., #6) in jSQL$_e$, there is no equivalent, separate operator in other systems. For MySQL, the system ran out of memory at #25, so no results are available after that.

#	Build Time (ms)			Space Cost (KB)			
	jSQL$_e$	PostgreSQL	MySQL	jSQL$_e$	Spark	PostgreSQL	MySQL
1	-	-	-	1,024,308	1,039,258	1,938,432	1,038,072
2	5,285	3,870	1,214	61	5	8	124
3	1	10	8	2	39	8	124
4	4,010	2,620	1,234	1	5	8	124
5	1	10	10	0	2	8	124
6	31,893	-	-	241,666	-	-	-
7	3,505	72,800	60,104	164,982	503,501	1,404,984	868,770
8	3,763	3,943	2,582	16,384	71,475	197,384	122,037
9	6,404	2,590	1,188	1	1	8	124
10	4,040	2,601	1,236	1	3	8	124
11	0	10	12	1	1	8	124
12	4	36,426	34,631	30	2,202,010	2,463,576	1,557,110
13	127,194	-	-	180,226	-	-	-
14	3,379	88,326	49,570	70,036	231,322	688,504	460,988
15	7,210	11,916	15,151	53,258	433,357	991,480	626,695
16	41,764	-	-	65,537	-	-	-
17	6,527	29,684	14,720	34,994	65,536	173,392	99,024

#	Build Time (ms)			Space Cost (KB)			
	jSQL$_e$	PostgreSQL	MySQL	jSQL$_e$	Spark	PostgreSQL	MySQL
18	0	-	-	3	-	-	-
19	0	2,121	1,458	2	71,475	193,256	118,786
20	8,757	13,052	15,448	53,249	854,938	976,864	617,441
21	2	10,474	10,053	6	515,584	976,864	617,441
22	16,636	-	-	98,305	-	-	-
23	5,895	66,027	64,173	53,251	602,624	960,824	607,313
24	40,811	-	-	65,537	-	-	-
25	9,380	35,001	14,768	43,862	84,173	172,144	99,024
26	91,750	34,602	-	90,129	576,922	1,212,744	-
27	26,545	-	-	57,345	-	-	-
28	8,600	19,692	-	23,907	83,968	163,216	-
29	0	1,711	-	2	63,488	128,080	-
30	2,623	222	-	124	805	1,616	-
31	34	15	-	18	265	240	-
32	33	16	-	18	265	240	-
33	113	16	-	39	418	344	-
34	0	19	-	1	264	240	-
35	68	15	-	18	231	240	-
36	19,392	-	-	98,305	-	-	-
37	5,893	85,595	-	53,248	602,624	960,824	-
38	50,720	-	-	57,345	-	-	-
39	9,298	48,940	-	16,063	29,491	62,432	-

#	Build Time (ms)			Space Cost (KB)			
	jSQL$_e$	PostgreSQL	MySQL	jSQL$_e$	Spark	PostgreSQL	MySQL
40	77,096	39,473	-	90,117	516,198	1,204,656	-
41	22,229	-	-	49,153	-	-	-
42	7,527	38,837	-	8,624	28,672	60,656	-
43	0	716	-	1	17,613	47,752	-
44	4,244	13,970	-	90,112	738,304	1,692,216	-
45	102,512	-	-	98,305	-	-	-
46	11,781	102,159	-	15,296	26,931	69,520	-
47	0	-	-	3	-	-	-
48	0	941	-	2	49,254	78,200	-
49	3,477	241	-	1	51	8	-
50	1,027	287	-	1,803	9,830	23,520	-
51	9,283	545	-	4	66	8	-
52	0	12	-	1	48	8	-
53	1	10	-	1	58	8	-
54	4,218	7,849	-	40,960	383,283	771,360	-
55	24,617	-	-	40,969	-	-	-
56	1	2,177	-	14	139	96	-
57	0	19	-	2	167	112	-
58	14	-	-	9	-	-	-
59	2	18	-	2	2	8	-
60	0	16	-	1	2	8	-
61	0	16	-	0	1	8	-

#	Build Time (ms)			Space Cost (KB)			
	jSQL$_e$	PostgreSQL	MySQL	jSQL$_e$	Spark	PostgreSQL	MySQL
62	788	3,261	-	16,385	107,213	237,280	-
63	7,820	-	-	16,387	-	-	-
64	1	-	-	5	-	-	-
65	0	679	-	1	127	32	-
66	3	18	-	3	42	32	-
67	988	5,675	-	40,961	325,734	745,328	-
68	0	9,572	-	4	335,565	822,160	-
69	139,938	31,526	-	81,925	349,491	1,060,720	-
70	0	9,976	-	6	359,322	1,136,480	-
71	168,591	-	-	40,967	-	-	-
72	2	2,751	-	11	136	72	-
73	0	19	-	2	165	88	-
74	24	-	-	7	-	-	-
75	3	-	-	1	-	-	-
76	0	22	-	1	6	8	-
77	0	24	-	0	6	8	-
78	727	3,164	-	16,385	-	237,280	-
79	0	2,892	-	4	110,285	261,744	-
80	93,285	12,270	-	32,773	117,862	337,704	-
81	0	3,322	-	6	120,934	361,824	-
82	51,610	-	-	16,387	-	-	-
83	1	-	-	5	-	-	-

#	Build Time (ms)			Space Cost (KB)			
	jSQL$_e$	PostgreSQL	MySQL	jSQL$_e$	Spark	PostgreSQL	MySQL
84	0	866	-	1	126	32	-
85	8	11	-	3	41	32	-
86	97,043	30,605	-	81,925	358,093	1,281,224	-
87	0	11,889	-	6	367,923	1,281,224	-
88	109,349	-	-	40,966	-	-	-
89	1	2,837	-	9	134	64	-
90	0	23	-	2	163	72	-
91	12	-	-	6	-	-	-
92	2	-	-	1	-	-	-
93	0	21	-	1	6	8	-
94	0	19	-	0	6	8	-
95	39,694	11,075	-	32,773	114,995	408,112	-
96	0	5,555	-	6	118,170	408,112	-
97	32,234	-	-	16,387	-	-	-
98	1	-	-	4	-	-	-
99	0	2,811	-	1	122	32	-
100	4	875	-	3	46	32	-
101	14,972	-	-	98,305	-	-	-
102	3,877	29,067	-	111,596	363,315	567,840	-
103	0	8,057	-	3	363,418	655,520	-
104	50,934	-	-	57,345	-	-	-
105	6,810	47,890	-	82,607	123,494	225,936	-

#	Build Time (ms)			Space Cost (KB)			
	jSQL$_e$	PostgreSQL	MySQL	jSQL$_e$	Spark	PostgreSQL	MySQL
106	159,818	31,896	-	90,117	684,442	1,370,088	-
107	58,398	-	-	57,345	-	-	-
108	12,674	34,089	-	42,698	117,043	210,024	-
109	0	2,362	-	2	58,163	147,856	-
110	0	7,967	-	4	363,622	742,928	-
111	65,201	-	-	49,153	-	-	-
112	6,351	49,066	-	30,120	48,845	80,672	-
113	144,765	36,613	-	90,117	621,056	1,358,072	-
114	43,985	-	-	49,153	-	-	-
115	11,759	51,554	-	15,242	45,466	78,344	-
116	0	885	-	1	23,552	47,752	-
117	557	7,096	-	40,960	334,438	760,664	-
118	11,920	-	-	77,825	-	-	-
119	3,215	22,431	-	86,626	282,112	440,768	-
120	0	5,737	-	2	8,704	306,200	-
121	0	4,504	-	1	17,408	306,200	-
122	86,491	-	-	36,866	-	-	-
123	1	-	-	4	-	-	-
124	0	1,612	-	1	80	24	-
125	4	25	-	2	87	24	-
126	180	2,154	-	16,384	107,520	237,280	-
127	3,340	-	-	28,673	-	-	-

#	Build Time (ms)			Space Cost (KB)			
	jSQL$_e$	PostgreSQL	MySQL	jSQL$_e$	Spark	PostgreSQL	MySQL
128	1,001	6,275	-	27,413	93,389	139,488	-
129	0	2,157	-	2	2,765	96,904	-
130	0	1,402	-	1	5,530	96,904	-
131	26,798	-	-	12,289	-	-	-
132	0	-	-	2	-	-	-
133	0	526	-	1	57	16	-
134	1	18	-	1	81	16	-
135	0	8,042	-	3	282,317	508,824	-
136	197,049	22,236	-	73,732	309,658	784,536	-
137	0	7,973	-	5	318,259	784,536	-
138	194,065	-	-	36,869	-	-	-
139	3	2,050	-	7	76	48	-
140	0	18	-	2	121	56	-
141	17	-	-	7	-	-	-
142	4	-	-	6	-	-	-
143	0	22	-	1	80	24	-
144	10	21	-	2	88	24	-
145	0	1,993	-	3	93,491	161,024	-
146	117,138	8,965	-	24,580	103,526	248,280	-
147	0	2,536	-	5	106,189	248,280	-
148	59,623	-	-	12,291	-	-	-
149	2	662	-	4	68	24	-

#	Build Time (ms)			Space Cost (KB)			
	jSQL$_e$	PostgreSQL	MySQL	jSQL$_e$	Spark	PostgreSQL	MySQL
150	0	27	-	2	107	32	-
151	7	-	-	4	-	-	-
152	2	-	-	4	-	-	-
153	0	28	-	1	65	16	-
154	5	24	-	1	84	16	-
155	0	6,133	-	4	282,522	576,672	-
156	178,047	20,859	-	73,732	237,773	712,864	-
157	0	9,057	-	5	246,374	785,776	-
158	156,317	-	-	36,869	-	-	-
159	3	2,163	-	7	77	48	-
160	0	25	-	2	123	56	-
161	13	-	-	7	-	-	-
162	3	-	-	6	-	-	-
163	0	33	-	1	80	24	-
164	8	24	-	2	88	24	-
165	0	1,931	-	4	93,594	182,496	-
166	74,155	7,264	-	24,580	74,957	225,624	-
167	0	2,618	-	5	77,722	248,704	-
168	47,359	-	-	12,291	-	-	-
169	1	703	-	4	66	24	-
170	0	28	-	2	104	32	-
171	6	-	-	4	-	-	-

#	Build Time (ms)			Space Cost (KB)			
	jSQL$_e$	PostgreSQL	MySQL	jSQL$_e$	Spark	PostgreSQL	MySQL
172	1	-	-	4	-	-	-
173	0	21	-	1	64	16	-
174	4	26	-	1	83	16	-
175	109,896	7,069	-	32,771	69,222	271,056	-
176	7,482	357	-	1	98	8	-
177	1	14	-	1	58	8	-
178	1	13	-	1	70	8	-

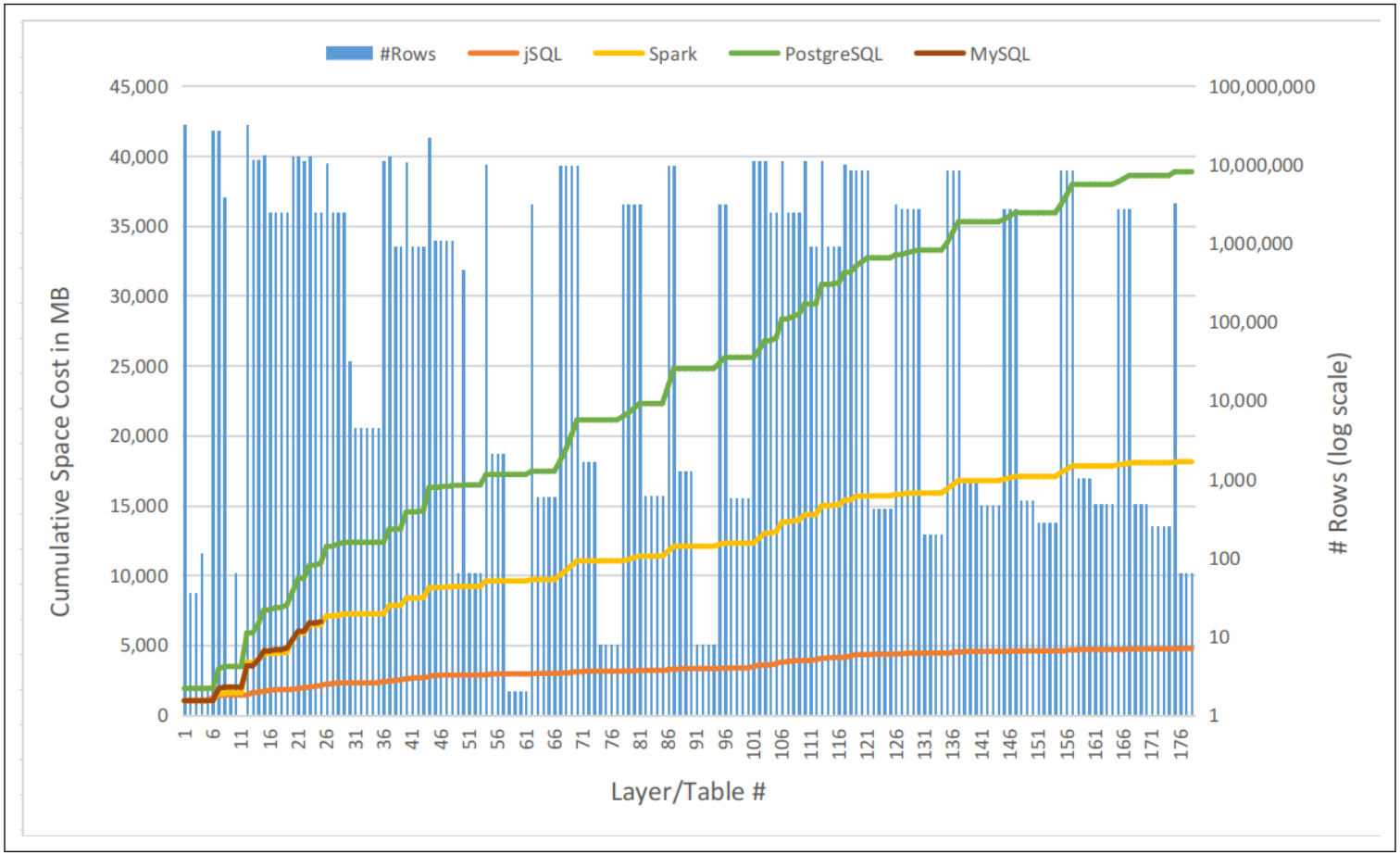


Figure 7-6: An illustration of the cumulative space cost in all four system that we tested as the data analysis progresses. The secondary (log scale) y-axis on the right shows the number of rows resulting from each step (the creation of a data layer or a table). Note that MySQL ran out of memory after Step #25. For more information, see Tables 7.3 and 7.4

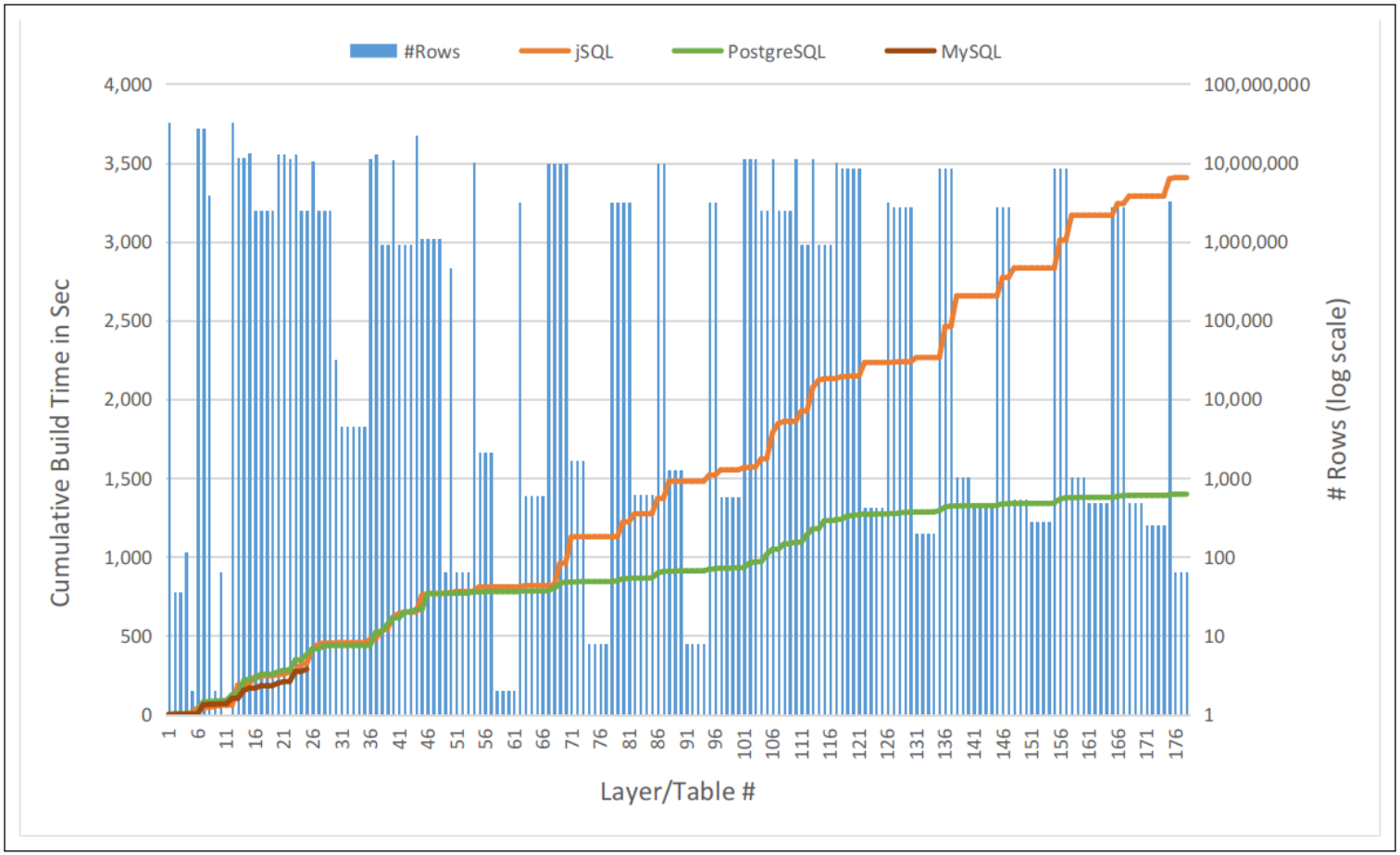


Figure 7-7: An illustration of the cumulative build time in all four system that we tested as the data analysis progresses. The secondary (log scale) y-axis on the right shows the number of rows resulting from each step (the creation of a data layer or a table). Note that MySQL ran out of memory after Step #25. For more information, see Tables 7.3 and 7.4

Table 7.5: The build-time results of running the min-max query on all 28 stacks. Each stack represents a statement (STMT, see Table 7.3). The first query (min_max_query0) was run on the original data set. Each of the remaining queries (1 to 27) was run on the layer/table at the top of the stack (the last step in each STMT), as illustrated by the column Input Layer #. The #Rows column shows the number of rows available at the top of the stack. The columns Stack Height, #DR layer, and #SB Layer are only relevant to jSQL$_e$ because for the other systems, the data is cached at the input table. For Spark, we tested two settings, one where Spark is allowed to store data only in memory, and the other where Spark is allowed to store data on disk if no memory is available. Note that MySQL ran out of memory after data analysis Step #25 and, therefore, we were only able to test min-max queries up to Query #8.

Query	Input Layer #	#Rows	Stack Height	#DR Layers	#SB Layers	Build Time (ms)				
						jSQL$_e$	Spark$_m$	Spark$_{m+d}$	PostgreSQL	MySQL
min_max_query0	1	32,950,296	1	0	1	2,068	134,233	136,515	10,090	4,038
min_max_query1	3	36	3	1	2	0	856	881	0	0
min_max_query2	5	2	3	2	1	0	2,324	2,286	0	0
min_max_query3	8	3,873,624	4	1	3	1,037	96,620	96,702	471	475
min_max_query4	9	2	2	1	1	0	264	253	0	8
min_max_query5	11	1	3	2	1	0	654	651	0	0
min_max_query6	14	11,704,508	4	1	3	5,341	187,082	207,819	1,494	0
min_max_query7	19	2,513,507	7	4	3	996	70,167	79,895	373	827
min_max_query8	29	2,513,467	12	5	7	2,704	334,801	366,688	311	186
min_max_query9	35	4,566	17	9	9	11	9,598	9,697	1	-
min_max_query10	43	937,079	12	5	7	934	244,190	254,109	123	-
min_max_query11	48	1,098,569	7	4	3	428	346,331	115,701	172	-
min_max_query12	53	65	16	10	15	1	21,066	11,473	0	-
min_max_query13	61	2	10	4	6	1	36,227	40,585	0	-

Query	Input # Layer	#Rows	Stack Height	#DR Layers	#SB Layers	Build Time (ms)				
						jSQL_e	Spark_m	Spark_{m+d}	PostgreSQL	MySQL
min_max_query14	66	611	8	4	4	1	14,590	15,979	1	-
min_max_query15	77	8	21	11	13	0	200,306	220,310	0	-
min_max_query16	85	613	18	10	11	2	61,308	66,692	0	-
min_max_query17	94	8	16	10	7	1	131,718	137,319	0	-
min_max_query18	100	577	13	9	7	2	43,974	48,148	0	-
min_max_query19	109	2,513,467	12	4	8	3,093	541,602	275,761	290	-
min_max_query20	116	937,079	13	5	8	1,108	182,217	193,744	109	-
min_max_query21	125	431	12	6	6	1	312,959	109,762	2	-
min_max_query22	134	201	13	7	6	0	45,991	39,890	0	-
min_max_query23	144	479	21	10	16	3	150,396	153,479	0	-
min_max_query24	154	279	21	11	16	2	58,829	62,869	0	-
min_max_query25	164	484	22	12	16	2	141,827	147,177	0	-
min_max_query26	174	258	22	13	16	1	57,066	63,770	0	-
min_max_query27	178	65	17	8	15	0	30,295	31,386	0	-

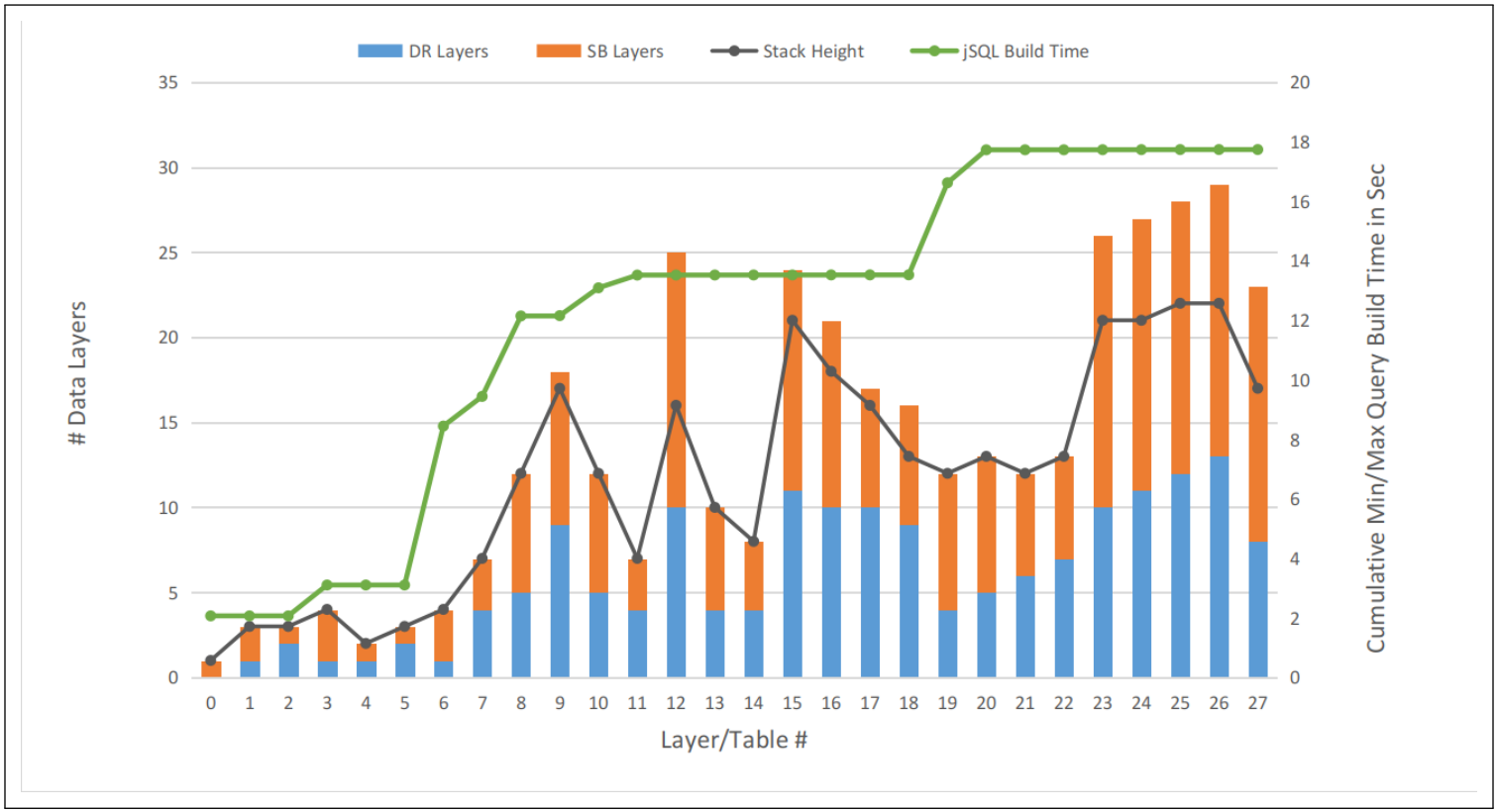


Figure 7-8: Illustrates jSQL$_e$'s cumulative build time (y-axis on the right) for the 28 (0 to 27) min-max queries listed in Table 7.5. The main y-axis (on the left) shows the number of layers at the top of the stack where the mix-max query was executed. In terms of the number of layers, we show the stack height, the number of SB layers in the stack, and the number of DR layers in the stack.

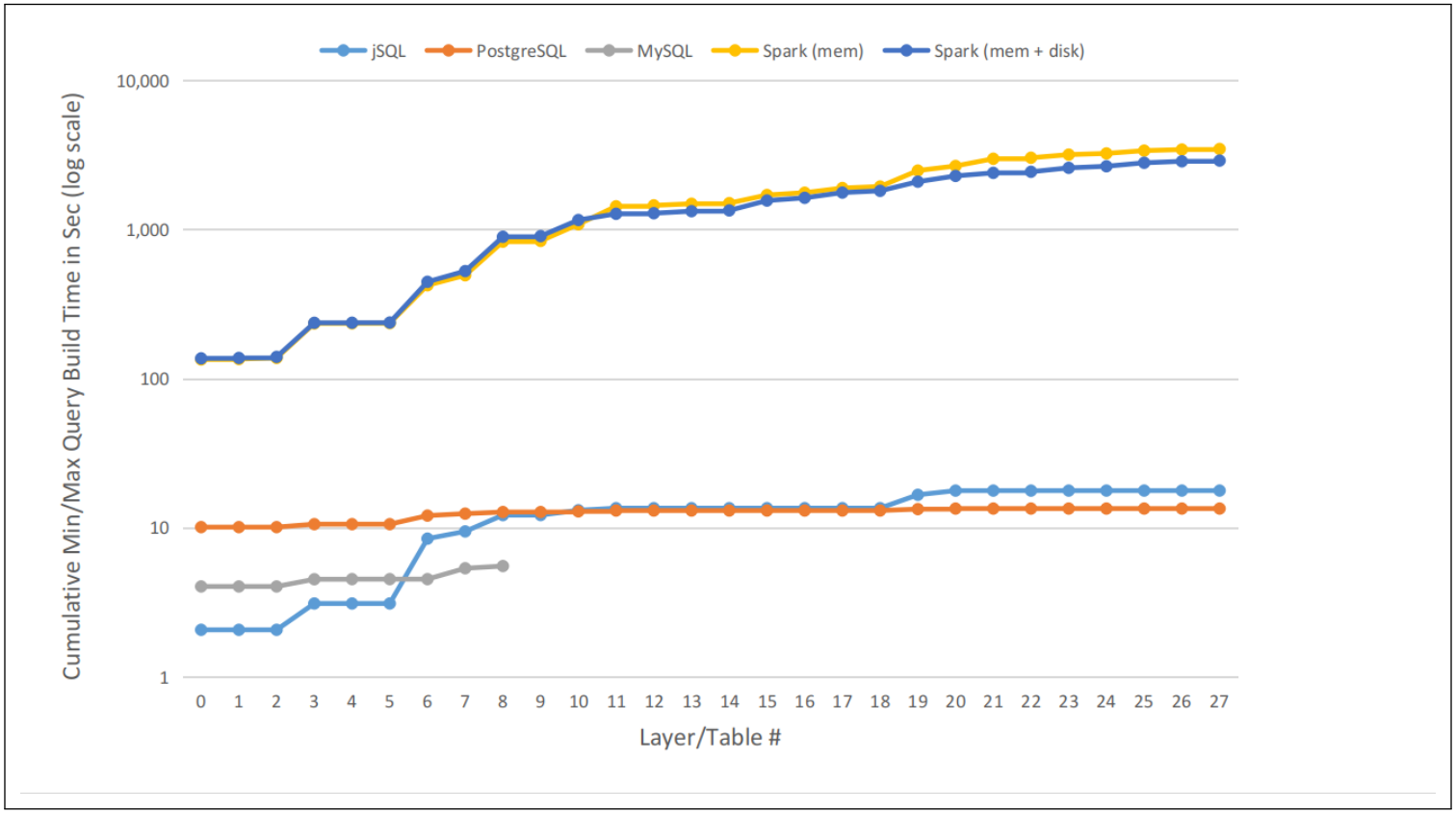


Figure 7-9: Illustrates the cumulative-build-time (only the top layer in each stack) comparison between all four systems for the min-max queries listed in Table 7.5. For Spark, we tested two settings, one where Spark is allowed to store data only in memory, and the other where Spark is allowed to store data on disk if no memory is available. Note that MySQL ran out of memory after data analysis Step #25 and, therefore, we were only able to test min-max queries up to Query #8. Also note that Spark does lazy evaluation, so all operators (in addition to the min-max query) are executed at the time of running the min-max query, hence the high build-time cost.

Table 7.6: Statistics about the data operators that were used during the data analysis in jSQL$_e$. The Count is the number of times the operator was used. The last column shows the total space-cost of using the operator. Note that a biggest cost is the GROUP operator, which none of the other systems support.

Operator	Count	Total Space Cost (MB)
AGGREGATE	34	756.02
AGGREGATE REF	2	104.00
DISTINCT	2	0.00
GROUP	36	1,708.11
IMPORT	1	0.00
JOIN INNER	13	624.09
JOIN LEFT	2	176.02
ORDER	16	0.04
PROJECT	52	0.15
SELECT	20	378.00

Table 7.7: The total space cost of all four systems. Note that for Spark, we tested two settings, one where Spark is allowed to store data only in memory, and the other where Spark is allowed to store data on disk if no memory is available. Also note that MySQL ran out of memory long before the analysis was over.

System	Storage Place	Total Space Cost (MB)
jSQL	all in memory	4,747
jSQL minus group operators	all in memory	3,039
Spark (mem only)	only ˜3GB in memory, the rest is discarded	18,118
Spark (mem + disk)	˜3GB in memory, the rest is on disk	18,118
PostgreSQL	all on disk	38,872
MySQL (Ran out of memory at Step #25)	all in memory	6,673

7.4.3 Discussion

In terms of space cost, there is no question about the superiority of jSQL$_e$ over the other systems. These results were not surprising, in fact, they were very much expected. The only question was, how far off the other systems would be from jSQL$_e$. From Table 7.7, we can see that the difference is significant and the techniques we

Table 7.8: The total build time (all steps) for all four systems. Note that for Spark, we tested two settings, one where Spark is allowed to store data only in memory, and the other where Spark is allowed to store data on disk if no memory is available. Also note that MySQL ran out of memory long before the analysis was over.

System	Total Build Time (min)
jSQL	57
Spark (mem only)	58
Spark (mem + disk)	48
PostgreSQL	24
MySQL (ran out of memory at Step #25)	5

used to save space were quite effective. However, the important question is, are these space savings worth the time cost. Since build time is ultimately what matters to the user, we have to judge jSQL$_e$'s time performance based on build time.

As we mentioned earlier, we spent only three months optimizing the core algorithms of our operators, which is not nearly enough time to get the system to optimum levels. There is a lot of room for improvements to reduce build time. But, even if we assume that jSQL$_e$'s build time cannot be optimized any further than it currently is unless we eliminate the dereferencing cost, we believe that jSQL$_e$ would still come out on top, and by a large margin. First, we can eliminate Spark since build time was more or less the same as jSQL$_e$, while the space cost was significantly larger than jSQL$_e$. As for MySQL, there is no point of using a system that can perform only 25 steps out of 178 steps that are required for the data analysis. The fact that jSQL$_e$ completed the analysis is enough for jSQL$_e$ to win over MySQL. So the only system that we need to talk about is PostgreSQL.

The main advantage that PostgreSQL has over jSQL$_e$ that significantly contributed to the fast build time is materializing the results. That is, at each step, the input data is immediately available (no extra steps or computations are needed to get the data) to the operator. On the other hand jSQL$_e$, has to dereference data blocks to reach the data. However, we know from the first experiment that only SB layers increase the dereferencing cost. We also know from this experiment that using

disk only affects build time slightly. So if jSQL$_e$ materializes the results of every SB layer and continues to use the space-saving techniques for DR layers, jSQL$_e$, for the use-case that we did, would need a total of 9GB[5], including the results of the `group` operators. In other words, we can still get the same build-time performance as PostgreSQL but with far less space cost if we materialize SB layers and use disk as a fallback if memory becomes full.

Obviously, there are a lot of variations between use-cases, and each system will behave differently with each use-case. Spark, given the right circumstances might be faster than PostgreSQL, or MySQL might be able to keep all the data in memory. The original data set might also be small, in which case it does not really matter which system we use; they will all perform well in terms of space and time. However, there are key differences that distinguish the systems regardless of what use-case that is being analyzed. Spark is designed to be as an in-memory data-analysis system that works in a distributed, server-based environment. PostgreSQL and MySQL are both relational database management systems that are designed mainly for a server-based environment. Although MySQL provides in-memory tables, those tables are not meant for permanent storage nor are they meant for storing large amounts of data. The common denominator between these three systems is that the user must have the technical expertise to know how to better use and utilize each system, something a typical analyst usually does not have. With jSQL$_e$, the system is specifically designed for a client-based environment. The space-saving techniques free the user from worrying about the technical side and allow the user to focus on the data-analysis side. For example, the user does not have to worry about how to conserve space and whether the memory is full or not.

To be fair, we did not use these systems (except jSQL$_e$) the way they were intended to be used. But that is precisely the point. The way these systems were intended to

[5]The 9GB can be computed using Spark's space cost since the storage cost of materialized data in Spark is the closest to that of jSQL$_e$ using the new customized storage engine. Simply take the overall space cost for jSQL$_e$ (4.6GB), subtract the total jSQL$_e$ cost of SB layers (1.5GB, excluding the `group`s), then add the total Spark cost of all the SB layers (5.9GB, excluding the `group`s).

be used does not fit the exploratory-analysis data model. So in reality, when analysts use these systems, they have to overcome many difficulties to achieve their goals, which is exactly what we did in this experiment (maybe to more extreme than what a typical analyst would do). We designed jSQL$_e$ from the ground-up specifically to fit the exploratory-analysis data model. So no more working against the current.

7.5 SUMMARY

In this chapter we discussed three experiments that were designed to test the effectiveness of the concepts and the techniques that we introduced in this research. In the first experiment, we focused on measuring the space cost and the access-time cost of using data-block references and DLIs to store intermediate results. The experiment was designed to eliminate other factors that could contribute to the space and access-time cost. The first question that we set to answer with this experiment was: How effective are the space-saving techniques compared to materialization? The experiment showed that the use of data-block references significantly reduced the space cost. The second question that we set to answer was: How effective are DLIs in reducing the dereferencing cost? The experiment showed that the use of DLIs maintained a constant dereferencing cost for operators with DR implementations with virtually no additional space cost.

The second experiment was to measure the efficiency of the new storage engine versus the old one. The first question that we set to answer was: How much space do we save using the new storage compare to the old one? The experiment showed that the structure of the new storage was up to 80% more efficient than the old one, especially for big data sets. Although the space-cost results were not a surprise, it was not clear whether the new engine would be faster than the old one in terms of access time. So the second question we set to answer was: How efficient is data-access time using the new storage compared to the old one? The results showed that the new engine's access time was more or less the same as the old one.

The final experiment was all about how jSQL$_e$ would perform in a real use-case. The question that we set to answer with this experiment was: How would jSQL$_e$ compares to other similar data-analysis systems in terms of space cost and build time? Although we only spent three months optimizing our prototype system jSQL$_e$, the results showed that jSQL$_e$ was significantly more efficient than any other system in terms of space and was comparable to the other systems in terms of build time.

What these experiments in total show is that, with careful design, we can provide users with a much better experience for exploratory data analysis. Our prototype, jSQL$_e$, provides a proof of concept that we can build an environment where multiple data-analysis tools can cooperate and share data without moving the data across these tools. The key idea to enable such cooperation and data sharing is keeping intermediate results around. The concepts that we introduced in this research provide a very cheap and relatively fast way to keep intermediate results in memory using a typical desktop or a laptop, without compromising on access time or build time.

CHAPTER 8: EXTENDING THE SET OF OPERATORS

In previous chapters, we discussed the concepts and the algorithms that allow us to keep all or most of intermediate results in main memory efficiently. However we only discussed seven main data operators. In this chapter we briefly talk about other operators that we implemented and also about various techniques that we can use to extend the set of operators that we can use in our shared data-manipulation system.

8.1 IMPLEMENTING OTHER OPERATORS

The following is brief discussion about other operators that we have implemented.

8.1.1 Other Types of Join

In addition to the vanilla inner `join` that we discussed in previous chapters, we also implemented other types of join. Although `cross join` has a bad reputation in terms of performance and in terms of the data that it generates, `cross join` within our data model costs virtually no space with no extra dereferencing cost compared to using the `join`-like algorithm. Since each record in the first input layer joins with all the records in the second input layer, we use equations to build the row and the column maps; that is, we can tell how to dereference a row i and a column j at the `cross join` layer using simple expressions instead of using explicit maps (arrays or lists). For example, row i in $\texttt{L}_{out}$ (the `cross join` layer) corresponds to row $floor(i/\texttt{L}_{in2}\texttt{.size()})$ in $\texttt{L}_{in1}$ and to row $i \ mod \ \texttt{L}_{in2}\texttt{.size()}$ in $\texttt{L}_{in2}$. However, similar to vanilla `join`, the dereference-chaining process still has to stop by the data layer to know where to go next (an SB implementation).

We also implemented the outer join operators (`left`, `right`, and `full`). The implementation is similar to vanilla `join` but, obviously, with a slight change to the core join algorithm. The final join we implemented is `semi-join`. However, it was easy to come up with a DR implementation for `semi-join`. Since `semi-join` projects only the first input-layer's columns, we can first reorder the resulting row indexes relative to the input layer. Then, we simply use the `select` algorithm to create the DLI from the first input layer's DLI. The reason we need to reorder the row indexes first is so that the indexes align with their respective DLs in the input layer's DLI, which makes it much easier and much faster to compute the output layer's DLI.

8.1.2 The `Distinct` Operator

We were also able to come up with a DR implementation for the `distinct` operator. From a logical perspective, `distinct` is a `group` followed by a `project` (to project away the group column). From an implementation perspective, we follow the same core algorithm for `group` to find the groups, but we keep only one record index from each group. We then reorder the record indexes relative to the input layer and then follow a mixture of the `select` and the `project` algorithms to build the DLI. Since `distinct` is a `group` followed by a `project`, we can optimize the algorithm to recognize a special case where we already have a `group` data layer on the same columns on which the `distinct` operator is applied. In such a case, we can simply avoid the grouping step and just perform a `project` to project away the group column.

8.1.3 Calculated Columns in the `Project` Operator

In addition to projecting existing columns from the input layer, `project` can also generate new columns by computing their values using expressions. Expressions can involve using values from existing column (e.g., `concat(first_name, ' ' ,last_name)`) or otherwise (e.g., `(1 + 1)` or calling a function `current_date()`). Usually generating a column in a data layer means that the operator's implementation (based on our

definition of a DL) becomes SB. However, we were able to extend our definition of DLs slightly to include expression maps in addition to row and column maps. This extension allowed us to propagate and combine expressions from the input layers to the output layer, while maintaining a column map for the columns that are being used in the expressions.

The extension allowed us to keep `project` with a DR implementation even when calculated columns are used. However, accessing data now (calling the `getValue()` function) requires evaluating the expressions, if any. We still believe that evaluating expressions on the fly in this case is much better than materializing the results of a `project` operator if calculated columns are used. Moreover, we do cache the results of the `getValue()` function that gets called on a specific column for the row that is being inspected[1]. This caching avoids recomputing expressions and paying dereferencing costs more than once if the same column is used multiple times in an expression or a statement.

8.1.4 The `Aggregate Ref` Operator

The `aggregate-ref` operator is like `aggregate`, but instead of returning the aggregation values for a function, it returns references to the rows that satisfy the aggregation function. For example. if we want to find the the minimum value in each group, we use `aggregate`, but if we want to find the row that contains the minimum value, we can use `aggregate-ref`. Using `aggregate-ref` in this case is analogous to $argmin()$ (or $argmax()$ for maximum values) in mathematics. Although we can achieve similar results using `aggregate` and `select`, such an operation consumes more resources (space and time) than is needed. The `aggregate-ref` operator eliminates unnecessary computations and uses less space because we do not have to cache any results. Moreover, we were able to come up with a DR implementation for the operator. Since the operator returns record references, we can reorder the records based on their index

[1]Once we move on to the next row, the cache is reset. In other words, we are materializing only one row and only the columns that are being used by the expression.

relative to the data layer from which the group records came. Then, we can simply follow an algorithm similar to the `select` operator to build the DLI.

Notice that not all aggregation functions can be used in with `aggregate-ref`, only those that return values from individual records. For example, it does not make sense to return row references for the `avg` function, but it makes sense to return row references for functions such as `min`, `max`, or `median` (in the case of an even number of samples, we can return both records in the middle).

8.2 METHODS TO EXTEND DATA OPERATORS

Up to this point, the only method that we have discussed to add new data operators to our data model is to come up with either an SB or a DR implementation for the operator. However, there are other methods that we can use to extend the set of operators. In this section we talk about some of those methods that we have explored.

8.2.1 Operator Composition

Many high-level data-manipulation operators can be constructed from a composition of more basic operators. For example, a `having` operator is a composition of an `aggregate` followed by a `select`. Instead of implementing such composable operators from scratch, we can simply take advantage of the implementation of existing operators and their space and time optimizations by wrapping the composition of operators in a virtual operator, similar to views in SQL. In other words, the virtual operator applies the composition of needed operators behind the scenes and stores the resulting data layers from each of the composed operators within a virtual data layer. The virtual data layer's schema is the schema of the final data layer resulting from the composition. In addition, data access requests from front-end applications or from data operators during build time can all be forwarded to the final layer. Although creating a customized implementation from scratch for these composable operators is probably more efficient (in terms of space, time, or both), as it is the case

with `aggregate-ref`, using composition is still far more efficient than caching data or running queries on the fly. Moreover, creating such compositions is a task that regular users can perform, like creating functions in R [32], and does not require a programmer to do it. Users can then reuse those compositions over and over in their data-manipulation environments. The question, however, can the user reuse layers within a virtual layer other than the final layer? The answer is, it is a design choice.

There is a trade-off that we have to make when we consider exposing the non-final layers in a virtual data layer versus not exposing them. Exposing the non-final layers means that we have to fully build the results of each layer even when the final layer does not need all the data from these layers. On the other hand, not exposing the non-final layers means we can optimize the composite operator as a whole by selecting the proper execution plan and by generating only the results that the final operator needs. Although the first option would increase reusability, we do not believe that it would be useful to a typical data analyst. The reason is that reusing a layer requires knowing the logic behind each layer and its results. A composite operator is supposed to be like a black box. So the logic behind each internal operator is unlikely to be apparent to a typical user, let alone knowing how to use the operators' results. Therefore, we believe that focusing on the second option would be far more beneficial to the user than the first one. We could also default to the second option, but if someone asked to see and use one of the internal layers, then we would fully build those layers.

8.2.2 Hybrid Implementations

So far, we have only discussed operator implementations that are either fully DR or fully SB. There is nothing that requires an implementation to be either one or the other. Although we have not implemented a hybrid operator, we believe we can have operators with hybrid implementations (to be explored in future work, but not part of this dissertation). That is, part of the data can be stored using a DLI, which does

not require a stop by the data layer itself, and the other part can be stored locally, which requires a stop by the data layer. For example, we can implement the `join` operator so that the block references from the first input layer are stored in a DLI (it is basically the `select` algorithm) and the block references from the second input layer are stored in a one-dimensional array. If the next operator uses fields from the first input layer, we can simply skip the `join` data layer, otherwise, we stop by the data layer. However, to enable operators with hybrid implementations to exist in our data model, we need to modify our definition of a DL slightly to allow columns to have different reference layers (`L'`). We still do not know exactly what that would look like, but, as we already mentioned, we intend to explore that in future work.

8.3 SUMMARY

We understand that the concepts and techniques that we discussed in this research to reduce the space cost of intermediate results require a careful design for each data operator. However, we believe that these concepts and techniques can extend readily to operators other than the ones we described in this research. In this chapter we talked about a number of other operators that we were able to implement either with DR or SB implementation. We also talked about the composition of operators as an easy approach to create additional operators. Although the goal is always to find a DR implementation for an operator, we might not be able to find one for many operators. We discussed a hybrid implementation which can take advantage of DLIs for parts of the data to reduce dereferencing cost. Next, we talk about related work (Chapter 9), and after that (Chapter 10), we discuss some future work and conclude this dissertation.

CHAPTER 9: RELATED WORK

Our research covers many aspects, and it is worth discussing some related work to our research for each aspect. In this chapter, we discuss related work for six aspects. The first and most obvious aspect is client-based data analysis (Section 9.1), given that our research is aimed towards facilitating data analysis in a client-based environment The second aspect is storing intermediate results (Section 9.2), given that our work focuses mainly on storing intermediate results efficiently. The third aspect is using data references (Section 9.3) in general, given that our block-referencing approach is a type of data references. The forth aspect is dataflow systems (Section 9.4), given that an SQL Graph can be seen as a dataflow structure. The fifth aspect is compression algorithms (Section 9.5), given that the main purpose of block references is to reduce the space cost of intermediate results by finding and reducing redundancy within SQL Graphs. The last aspect is model-based data management systems (Section 9.6), given that models are another very efficient way to store data, if the data fit certain criteria. Next we discuss the related work for each one of these six aspects.

9.1 CLIENT-BASED DATA ANALYSIS

There are many client-based data-analysis systems and tools that vary from the simple, straightforward spreadsheet to the complex, fully fledged database management system (DBMS). As we move across the spectrum, we see trade-offs between simplicity on one side and power and flexibility on the other. Systems such as spreadsheets, R [32], Matlab, SAS, Tableau [61], and Voyager [69] are one-stop data-analysis solutions that are relatively easy to use even by non-technical individuals. However,

these stand-alone systems offer predefined and limited data-analysis capabilities and they are difficult to integrate with other systems, such as external visualization tools, to expand their data-analysis capabilities. Moreover, inspecting their intermediate results (possibly by external tools) is either unsupported or available only through exporting and importing data from one system to another.

Moving along the spectrum, we start to see systems that specialize in a certain aspect of the data-analysis process, such as DBMSs, thus making them more powerful in performing a class of tasks. Data analysts can combine different systems specializing in different classes of tasks to build a data-analysis ecosystem, thus providing flexibility to data analysts to choose which system should perform which tasks. For example, DBMSs provide a variety of data-storage and data-manipulation capabilities. The analyst can choose a lightweight relational DBMS, such as Microsoft Access, or a more robust, heavyweight DBMS, such as PostgreSQL [28]. The analyst can also separate storage from data manipulation by choosing, for example, Hadoop's HDFS [60] to store big data and use Pig [50] or Hive [63] to manipulate the data. On the front-end side, the analyst can select from a variety of visualization tools, for example, to display the results. Tools such as Tableau [61], Zeppelin [6], and Jupyter [37] can pull data from DBMSs then build and render plots based on that data.

On the far end of the spectrum where we have the most flexibility, we see programming languages such as C/C++, Python [25], and Java [51]. In addition to the low-level functionality that programming languages provide, each language has its own set of high-level data-analysis libraries, each of which specializes in a certain aspect of the data-analysis process. For example, Python has libraries such as NumPy [49] for statistical data computations, Pandas [46] for manipulating data in a tabular form (tables), Matplotlib [31] for visualizations, TensorFlow [1] for large-scale machine learning, and many more. Although each library can be optimized to be highly efficient in terms of space and time, combining these libraries to perform

a complex data-analysis task can be highly inefficient because of data movement between the individual tools. Systems such as Weld [52] eliminate the data movement overhead by providing a runtime API. Instead of each library performing its own computations, libraries submit the code for the computations that they want to perform to the API using what is called an *intermediate representation* (IR). Once the IRs from the involved libraries are collected, the Weld runtime combines the code and performs cross-library optimizations and loop fusions, then compiles the code and runs it. The result is an executable code that is highly optimized specifically for the data-analysis task in question. However, Weld provides a shared environment for libraries only within a single application. Moreover, Weld does not keep intermediate results.

The more complex the data-analysis ecosystem becomes, the more we lose simplicity and the more complicated the integration process becomes among the individual components. Even if the data analyst has the technical knowledge and the skills to build and manage such a complex ecosystem, intermediate results are not easily accessible, making cooperation difficult between the individual components of the ecosystem. By using SQL Graphs, we are able to provide a data-analysis ecosystem core that factors out the data-manipulation process and maintains all or most intermediate results. This ecosystem core removes the complexity associated with moving data among the individual components and allows easy cooperation and data sharing.

9.2 STORING INTERMEDIATE RESULTS

Certain components within a data-analysis ecosystem manipulate data for various reasons. Some of those components allow their intermediate results to be inspected and shared either directly or indirectly, and other components do not; for those components that do, it is usually through indirect methods. For example, to inspect intermediate results of a query, say in a relational DBMS, we would have to run each operator separately and materialize the results, each in a separate table. Moreover,

it would not be a simple modification for relational DBMSs to support intermediate-result inspection because execution is pipelined; so full intermediate results do not exist at a particular point in time. In a Hadoop-based [60] data-manipulation system such as in Pig [50] and Hive [63], intermediate results must be written to files and flushed to disk if we want to inspect those results. The process is inefficient and expensive in terms of time and space.

Systems that allow storing and sharing intermediate results directly do so at the request of the user and without space-saving techniques, at least none that would have a big effect. For example, Spark [71] uses RDDs to store intermediate results and share them across applications. Users can choose which intermediate results they want to persist, thus allowing immediate data availability. However, to the best of our knowledge, Spark does not try to reduce the footprint of those intermediate results, at least not in a way that would make a difference. Although RDDs store lineage information, the information does not provide immediate data availability—it is only used to recompute and reconstruct the data if needed later. As a result, the user has to be strategic about which results he or she should persist based on the amount of memory available and the data-availability response time that the application needs.

Because the footprint of block references is so small compared to the otherwise materialized data, SQL Graphs can retain in main memory all or most intermediate results in data layers that can be shared across applications directly without having to involve the user. In addition, dynamic adaptations can be added to trade-off space for time or vice versa to make sure that the environment stays within the specified space and time limits.

9.3 USING DATA REFERENCES

Data references have long been used in data structures of all kinds. However, we are interested specifically in using data references during the data-manipulation process. In main-memory databases, Lehman and Carey [42] introduced a concept similar

to data layers called *temporary lists*. Since the database is in main memory, they concluded that it is more efficient to move tuple references between data operators instead of tuples of data. The intermediate results are held in temporary lists, which are special relations that consist of a *description* for the columns and a *list of tuples of references* to the actual data tuples. However, these lists are used internally and discarded once the query is processed and cannot be inspected. Unlike temporary lists, data layers have customized physical representations for each operator to maximize their space efficiency. Moreover, data layers keep their data in memory and can be inspected at any time.

Disk-based databases usually use pipelining [16, 22, 27] to move the data itself between operators instead of references (to data on disk) because of the high disk-access overhead. However, there have been certain cases where using references in such databases improved efficiency. Valduriez [65] introduced *join indices* as a mechanism to speed up joins when join selectivity is low. The index is a precomputed-join of on-disk references (referred to as surrogates [17,30]) to the original tuples that satisfy the join operation. The index is then used for similar join operations instead of recomputing the join. In contrast, operators in our data model do not use pipelining. The data itself does not move through the operators; instead, result references are calculated and stored at each data layer to provide immediate data availability.

9.4 DATAFLOW SYSTEMS

There are many dataflow systems that range between low-level general-purpose systems and high-level domain-specific systems. Low-level dataflow systems such as Hadoop map-reduce [5, 20], Dryad [35], and Haloop [12] provide great data-analysis flexibility, but they require programming experience and they are too complicated for many data analysts and domain experts to use and integrate with other systems. Moreover, these systems are designed for server-based and cluster-based data-analysis environments, which makes them even more difficult to integrate with other systems.

There are dataflow systems that offer in-memory data-analysis capability, such as Spark [71], which allows for data-set reuse across multiple jobs and offers much faster responses than disk-based systems. However, these systems still require programming experience to work with and they are difficult to integrate with other systems.

Other systems such as Pig [50], Hive [63], and SCOPE [13] provide a higher-level abstraction over some of the dataflow systems above. Pig users can build data-analysis plans relatively easily—as opposed to writing pure map-reduce jobs—using Pig Latin scripts which are then compiled and executed as map-reduce jobs. Pig also has a provenance-tracking framework called Lipstick [4] that users can use to query and track the execution process of their data analysis. Hive and SCOPE also provide a high-level abstraction using declarative, SQL-like languages to hide complex details from the user. Although these systems are much easier to use, the user still needs to have a level of programming experience to use them and integrate them with other systems because of their inherent dependency on other low-level dataflow systems.

Domain-specific dataflow systems provide the highest level of abstraction and perhaps the most suited for non-programmer users such as data analysts and domain experts. For our purpose, the word "domain" here means data-analysis techniques such as visualization, machine learning, and data sampling. Systems such as the Visualization Toolkit (VTK) [59], IBM's Visualization Data Explorer [3], and Reactive Vega [58] provide high-level abstractions to build data visualizations while hiding the technical details to convert data into visualizations. Reactive Vega, for example, uses Vega's declarative visualization grammar [41] to build the dataflow graph. Although the user requires far less programming experience to use these domain-specific systems for their intended domains, they still require a lot of technical and programming experience from the user to integrate with other domains and other systems. Moreover, such a high-level of abstraction tends to hurt the data-analysis process, such as disabling the user from examining the data manipulation process or the intermediate results that led to constructing, for example, the visualization.

Although we do not consider SQL Graphs as dataflow systems, they can be modified to function as a non-distributed client-based dataflow system and can provide great advantages over existing dataflow systems. Low-level dataflow systems such as Hadoop map-reduce [5, 20], Dryad [35], and Haloop [12] require the user to pre-build the execution plan before starting the execution process, then wait for the final result to be stored on disk. Such systems are slow and allow inspecting only the final result. Pig [50] has a tool called `illustrate` that allows inspecting intermediate results but only on a sample data, not the full data set. Moreover, `illustrate` manufactures data if no data passes through certain operations. In other words, `illustrate` is made for debugging purposes, not for data-analysis purposes. On the other hand, SQL Graphs reside in memory and allow inspecting intermediate results of full data sets.

Other in-memory systems such as Spark [71] allow the execution plan to be built progressively with much faster performance, while allowing intermediate-result inspection. However, persisting intermediate results is expensive, which forces the user to be strategic about which results to keep and which ones to recompute if needed. SQL Graphs allow execution plans to be built and executed progressively within interactive speed and allow intermediate results to persist in main memory with a small footprint. In addition, SQL Graphs shift the burden of integration from the tool user to the tool developer. In other words, the integration cost is paid once during the development of the data-analysis tool, as opposed to other dataflow systems where the user of the tool has to pay the integration cost every time he or she uses the tool.

9.5 COMPRESSION ALGORITHMS

Compression algorithms have long been used to reduce the size of data. The key idea behind these algorithms is finding redundancy in the data and replacing it with something smaller in size. Usually these algorithms do not compress individual data values, instead, they compress blocks of data, to have a much higher chance of finding redundancy. To access the data values inside the compressed data blocks, many

of these algorithms require decompressing the entire data block first, such as LZ [72], Huffman Coding [40], X-match [39], FVC [70], C-PACK [15], and many more. Although the compression computation can happen in the background, hiding the cost from the end-user, the decompression cost is difficult to hide because decompression is needed before accessing data values. Other algorithms, such as MILC [67], PforDelta [73], EF encoding [66], LZ trie [57], and phonebook databases [56], do not require decompressing entire data blocks. However, such algorithms are usually used only for special cases and are not suitable for general-purpose compression. For example, MILC and PforDelta are used for compressing inverted lists (ordered lists of integers).

There are many compression techniques [2, 10, 14, 24, 48, 68] that were introduced specifically for main memory settings, ranging from embedded systems to high-end servers. The main purpose of these algorithms, in addition to reducing the size of the data, is either to eliminate or reduce reliance on disk, which ultimately improves time performance. The compression and the decompression cost is usually much less than the cost of fetching the data from disk. Even if disk is used to store the compressed data, fetching the compressed data from disk to memory then decompressing it can still be faster than fetching the fully-decompressed data. However, for practical cases, most compression algorithms can achieve at most a 2× compression ratio [47], and a few [8, 9] can achieve a 3–4× compression ratio. Moreover, these algorithms are mainly designed for high-end-server environments where the main memory is large. In terms of storing intermediate results on a client-based environment, we need far more than a 2–4× compression ratio.

Although data-block referencing is not technically a general-purpose compression algorithm, within the context of storing intermediate results of data operators, the technique is general in a sense that it reduces the cost of storing the results regardless of the contents of the data. Moreover, data-block referencing does not require decompressing the results (materializing them) to access data values. Furthermore, the space cost of data-block references is small compared to the actual data blocks

that they reference. General-purpose compression algorithms can be used to further reduce the size of the working data sets, providing more space for data analysis. However, the implications on time performance are yet to be determined. Algorithms such as MILC [67], PforDelta [73], and EF encoding [66] can also be used to compress the row indexes inside many data layers, where the indexes are usually ordered lists of integers such as `select` data layers. These algorithms are especially useful because they reduce the space cost of many data layers without the need for decompressing entire data blocks to access data values.

9.6 MODEL-BASED DATA MANAGEMENT SYSTEMS

There are cases where storing the exact observed value is not necessary as long as the stored value is within a certain error boundary, such as renewable-energy sensors [36]. For such cases, the data, specifically time-series data, can be represented using a *model* instead of the actual values. For example, we can represent the observations within a time period using a linear equation where the variable represents the timestamp. The result is a data set with a fraction of the space cost that we would need to store the individual observations with their exact values. There are many methods [23,29,44,53,54] that have been proposed for building these models. In addition to the techniques for building the models, there are model-based data management systems, such as MauveDB [21], FunctionDB [62], Plato [38], Tristan [45], and ModelarDB [36]. Although model-based techniques are extremely efficient at saving space and time, they are useful only for data that can tolerate a certain margin of error and that does not fluctuate much.

Combining model-based techniques with data-block referencing, we can achieve much greater space savings for many classes of data sets in terms of the cost of storing the working data sets and the data layers themselves. Moreover, model-based techniques can provide big time saving. For example, performing many aggregations can be done by solving an equation, which is $O(1)$, regardless of the size of the data.

9.7 SUMMARY

Our work is not meant to replace any of the related work that we discussed in this chapter. For example, using jSQL$_e$ does not mean we abandon using DBMSs of all types nor does it mean to stop using front-end tools such as Tableau [61] or R [32]. Our work is meant to complete a missing link in the data-analysis ecosystem. For example, jSQL$_e$ can act as an intermediate layer between DBMSs and front-end tools or as a data-manipulation infrastructure to facilitate cooperation among various data-analysis tools. Space reduction techniques such as compression algorithms and model-based data storage, although not suitable for storing general-purpose intermediate results, they can be integrated with jSQL$_e$ to make it even more space-efficient.

CHAPTER 10: FUTURE WORK AND CONCLUSION

In this research we explored the problem of analyzing data in a client-based environment. The main issue that we focused on was inefficient data sharing across multiple data-analysis tools that is necessary to accomplish many data-analysis tasks. The sharing is usually achieved by moving data manually from one tool to another. As a solution, we introduced a new data paradigm and data model that allow front-end applications to share data and intermediate results without having the user move data back and forth between these applications. We introduced SQL Graphs and data-block referencing that allow us to efficiently store in main memory all or most intermediate results of typical data-analysis sessions on a personal computer or a laptop with an 8GB of RAM. We also introduced the concept of a DLI that allows us to keep data-access time within interactive speed, a requirement that many front-end applications need. We implemented jSQL$_e$, a prototype for a shared data-manipulation system using the concepts we introduced in this research.

Our experiments show that our system, despite us spending only three months optimizing it, was comparable to other well developed systems in terms of time. In terms of space, our system required 8-16% of the cost that is needed to store the data in the other systems. Such a significant reduction in space cost allows us to keep intermediate results in main memory, which in turns allows front-end applications to share these results without moving data across these applications. For the rest of this chapter, we briefly discuss some future work and research venues that can branch out of this research. Then we finally conclude this dissertation.

10.1 FUTURE WORK

We barely scratched the surface with SQL Graphs and data-block referencing. We believe that there is still a lot more to explore. The following are some of the interesting topics or research paths that we think are worth exploring.

10.1.1 Using Indexes

Indexes are used to speed up the data-lookup process. For large data and highly interactive applications, the build-time that we have achieved so far might be too slow. Using indexes can significantly reduce the build-time cost. However, creating indexes is not cheap in terms of space. In addition, there is no such a thing as a general index. That is, we cannot create one index and expect to use it for all possible types of lookups. So we need indexes that can take advantage of data-block referencing. For example, we should be able to use an index that we create on the input layer of a `select` operator for lookups that we do on the output layer. However, we somehow have to account for the rows that we filtered out.

The simplest way to account for the filtered rows is to reapply the `select` condition to the appropriate rows as we find them using the index. However, if there is a stack of layers between the layer where we want to use the index and the layer to which the index belongs, we have to reapply the operator of every layer in that stack on each row that we find, which is basically the pipelining approach that we see in most DBMSs. It is not clear whether pipelining would be the only approach or whether there could be a better approach. The point is, the use of block referencing creates opportunities for creating indexes that we believe are far more space-efficient than the traditional ones. The question is, how can we utilize them properly in other layers.

10.1.2 Hybrid Operator Implementations

In this research we discussed two types of implementations for data operators, SB and DR implementations. However, there is nothing that prevents us from creating a hybrid implementation. For example, in a `join` operator, there are two references for each record (in the output layer), one from each input layer. The implementation that we discussed in this research was an SB implementation. However, if we sort the records based on the references from, for example, the first input layer, we can use a DLI to store the references from the first input layer, and store the references from the second layer in a regular list. Now assume we apply a `project` on the join's output layer but we only project columns from the join's first input layer. In this case we can use the DLI and we do not have to stop by the join layer during the dereferencing process. In other words, the use of hybrid implementations can decrease the overall percentage of stops that we have to make during the dereferencing process.

10.1.3 Dynamic Materialization

During the data analysis process, the need for space versus speed can change depending on the stage and the application that will use the results. At the beginning of the data-analysis process, space might be more important because the data set is still large and the analyst is still exploring multiple paths. However, as the analyst zooms in on a certain path, the results become smaller and more manageable by the application, such as visualization tools, at which point the speed might become more important. One way we can provide faster data-access time is by materializing the data at the layer in question. Although we can allow the user to decide whether to materialize the data or not at at given data layer, we believe that making such decisions requires technical expertise beyond what a typical data analyst has. So the system should be able to dynamically decide whether the data at a given layer should be materialized or not based on simple parameters that the user can provide and can easily understand, such as space limit and interactive-speed limit.

10.1.4 Lazy Evaluation

Although we concluded from our experience with Spark [71] that pure lazy evaluation is not suitable for exploratory and interactive data analysis, there might still be situations where lazy evaluation is useful. We do not know yet what those situations might be. But in jSQL$_e$, we decoupled the execution of an operator from the returning of its results. That is, the user sends the request to the system to execute a certain operator, and instead of waiting for the results, the system immediately returns the id for the layer that will contain the results. Then later the user needs to send another request to ask for the data of a layer with the given id. This approach allows the system to execute the operators in the background while the user is constructing and thinking about the execution plan.

So the idea is that we do not want to wait until the user or the application wants to see the data to start the evaluation process for the entire data-layer stack. However, it might be more efficient, in terms of space and time, to wait for two to three layers along the stack before we start evaluating. We know that as we add more layers to the stack, the size of the results is more likely to get smaller. So, for example, if we wait to see what the next two operators that the user will apply are after a given layer, we might not have to perform a full evaluation on that layer and we might only need to process a subset of the data that is needed by the next two operators.

10.1.5 Extended Disk Storage

As we saw in Chapter 7, PostgreSQL [28], a disk-based DBMS, was surprisingly fast in terms of build and access time. There are two main reasons behind the fast performance. The first is data caching and data eviction policies. The short story is, the data is first loaded from disk into main memory (cached) one page at a time (or multiple pages at a time). Then, based on how often or how recent the data in a given page is used, eviction policies decide which page gets evicted from the main

memory back to disk[1] when the database exceeds the memory limit (the buffer size). The strategy of evicting the least-used pages (cold pages) from memory means that as long as we operate on data that is within the most-used pages (hot pages), the disk overhead will not be an issue. This observation brings us to the second reason, which is the nature of exploratory data analysis.

For the most part of the exploratory data-analysis process, our observation is that the analyst continues to operate and process the result from the previous operator. This behavior means that there is a high chance that the data we need for the next operator is in hot pages. We believe that with proper data-eviction policies, we can utilize disk storage to extend the capacity of SQL Graphs in a client-based environment. However, there is another big factor that contributed to PosgreSQL being overall faster than the jSQL$_e$ in our experiments: the inputs to the operators are materialized data, while the jSQL$_e$ has the extra dereferencing cost to reach the data. So we still need to see the effect that disk-storage support will have on the overall performance of the system. However, since we now have disk support, we have the option of materializing results more often than we would using only main memory. This option allows us to trade the dereferencing time cost for the materialization space cost and vice versa.

10.1.6 Data Compression

Although compressing data is expensive, especially if it requires decompression to access the data, we believe that there are still advantages to using data compression algorithms in certain cases. Using algorithms such as MILC [67], PforDelta [73], and EF encoding [66] can provide at least 40% reduction in the space cost of storing data-block references (see Section 9.5), especially since these algorithms do not require decompressing entire blocks of data to access the individual data values in these blocks. Also, using general-purpose compression algorithms to compress working data

[1] In the case where the data in the page to be evicted has not been modified, the page is just evicted from memory but not sent back to disk because the data is already on disk.

sets can significantly increase the space that we have left for data analysis. However, such compression algorithms must be combined with performance-enhancement techniques, such as caching decompressed portions of working data sets, for in-memory compression to reduce the decompression overhead. So we still have to figure out which algorithms work best and in what circumstances.

10.1.7 SQL Graphs in Distributed Environments

Since the beginning of this research, our aim was working in a client-based environment and keeping data in main memory. But there is no reason why the same concepts cannot work in a server-based environment. However, the concepts as described in this research are valid only for a single server. Since vertical scaling (in our case, increasing the memory of a server) has limits and is expensive, we turn our attention to horizontal scaling (using multiple servers to perform a task). There are many challenges that arise when we try to use these concepts in a distributed environment (where the system's functionality is distributed across multiple servers). The two main challenges are: 1) how to reference data blocks on different servers and 2) how to deal with network latency. The interesting thing about using a server-based environment is that we have more room in terms of time and a lot more room in terms of space. So we can make trade-offs in a server-based environment that we could not make in a client-based environment.

10.2 CONCLUSION

In the context of data analysis, there are many tools to choose from to perform a data-analysis task, varying from simple and limited to complex and flexible. Creating a monolithic system that can serve all data-analysis needs (present and future) is extremely difficult to impossible. A better approach is to embrace diversity and create a system that can facilitate the integration of these diverse tools and allow them to cooperate. However, these tools are largely disconnected, leaving the end-

user with the daunting manual task of moving data back and forth between these tools and performing data-format conversions. Users with less technical expertise opt for simple and straightforward tools to analyze the data, preventing them from unlocking the full potential of their data. Users with enough technical expertise can still spend a significant amount of their time on data movement and conversion, forcing them to opt for simple tools to reduce costs or to meet deadlines, for example. Even when time and cost are not an issue, there are still applications for which this environment (multiple data-analysis tools connected by manual data movements) is not suitable because it is too slow for the application needs—such as interactive visualization tools—without adversely affecting space.

In this research we explored a new data paradigm and data model (Chapter 2) in which data-analysis tools can share all or most of their intermediate results to eliminate data movement. We focused on client-based environments where resources, such as memory (RAM), are limited. Within this new paradigm, data-analysis tools relinquish data-manipulation tasks to a shared data-manipulation system where all or most intermediate results are kept in memory using SQL Graphs. Other tools can access these results at any time, and the data is immediately available. However, since memory capacity in typical client-based environments is small, using traditional methods was not feasible to store the large amount of data generated by those intermediate results. So we introduced an extremely efficient way to store intermediate results using data-block references (Chapter 3). We also introduced DLIs (Chapter 5) as an indexing mechanism to speed up data-access time.

To examine the effectiveness of the concepts that we introduced in this research, we implemented jSQL$_e$, a shared data-manipulation system. Testing the system (Chapter 7) with a simulated and controlled use-case showed that the concepts worked in practice as predicted by our theories. On the other hand, testing our system against other well established and well developed systems using a realistic use-case showed that our system was far superior in terms of space efficiency, while it was comparable

to the other systems in terms of time efficiency. Despite spending only three months on optimizations, the system already exceeded our expectations. We were able to keep in memory all intermediate results (178 results) of a realistic use-case over a large data set, in addition to keeping the data set itself in memory, with less than 6GB of storage. On the other hand, the other systems were able to keep only a fraction of those results.

As we mentioned in Section 10.1, we barely scratched the surface, and there is still a lot more to explore and many challenges to overcome. Adopting the new data paradigm that we presented in this research by front-end applications is also a challenge and will take time. But we hope that once this new shared data-manipulation system sees the light, more and more applications will start adopting the new paradigm, thus making data analysis easier and more accessible. Hopefully this research can bring us one step closer to unlocking the full potential of data.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[2] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith. Memory expansion technology (MXT): Software support and performance. *IBM Journal of Research and Development*, 45(2):287–301, 2001.

[3] Greg Abram and Lloyd Treinish. An extended data-flow architecture for data analysis and visualization. In *Proceedings of the 6th Conference on Visualization '95*, VIS '95, pages 263–270, Washington, DC, USA, 1995. IEEE Computer Society.

[4] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *Proc. VLDB Endow.*, 5(4):346–357, December 2011.

[5] Apache. Apache Hadoop, 2017. `http://hadoop.apache.org`.

[6] Apache. Apache Zeppelin, 2017. `https://zeppelin.apache.org/`.

[7] Apache. Apache Spark, 2020. `https://spark.apache.org/`.

[8] Angelos Arelakis and Per Stenström. A case for a value-aware cache. *IEEE Computer Architecture Letters*, 13(1):1–4, Jan 2014.

[9] Angelos Arelakis and Per Stenstrom. SC2: A statistical compression cache scheme. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 145–156, June 2014.

[10] L. Benini, D. Bruni, A. Macii, and E. Macii. Memory energy minimization by data compression: algorithms, architectures and implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):255–268, 2004.

[11] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, Dec 2011.

[12] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[13] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, August 2008.

[14] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Wolf. Energy savings through compression in embedded java environments. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, page 163–168, New York, NY, USA, 2002. Association for Computing Machinery.

[15] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas. C-Pack: A high-performance microprocessor cache compression algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(8):1196–1208, 2010.

[16] H-T. Chou, David J. Dewitt, Randy H. Katz, and Anthony C. Klug. Design and implementation of the Wisconsin Storage System. *Software: Practice and Experience*, 15(10):943–962, 1985.

[17] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, December 1979.

[18] Oracle Corporation. MySQL, 2019. `https://www.mysql.com/`.

[19] Douglas Crockford. JSON, 2019. `https://json.org/`.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[21] Amol Deshpande and Samuel Madden. MauveDB: Supporting model-based user views in database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 73–84, New York, NY, USA, 2006. Association for Computing Machinery.

[22] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 228–237, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[23] Frank Eichinger, Pavel Efros, Stamatis Karnouskos, and Klemens Böhm. A time-series compression technique and its application to the smart grid. *The VLDB Journal*, 24(2):193–218, 2015.

[24] M. Ekman and P. Stenstrom. A robust main-memory compression scheme. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 74–85, 2005.

[25] Python Software Foundation. Python, 2020. `https://www.python.org/`.

[26] Google. Google Maps, 2019. `https://www.google.com/maps`.

[27] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. *SIGMOD Rec.*, 19(2):102–111, May 1990.

[28] The PostgreSQL Global Development Group. PostgrSQL, 2019. `https://www.postgresql.org/`.

[29] Tian Guo, Zhixian Yan, and Karl Aberer. An adaptive approach for online segmentation of multi-dimensional mobile data. In *Proceedings of the Eleventh ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDE '12, page 7–14, New York, NY, USA, 2012. Association for Computing Machinery.

[30] Patrick Hall, John Owlett, and Stephen Todd. Relations and entities. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 201–220, 1976.

[31] John Hunter, Darren Dale, Eric Firing, Michael Droettboom, and Matplotlib development team. Matplotlib, 2020. `https://matplotlib.org/`.

[32] Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[33] SAS Institute Inc. SAS, 2019. `https://www.sas.com/`.

[34] The MathWorks Inc. Matlab, 2019. `https://www.mathworks.com/`.

[35] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007.

[36] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. ModelarDB: Modular model-based time series management with spark and cassandra. *Proc. VLDB Endow.*, 11(11):1688–1701, July 2018.

[37] Project Jupyter. Project Jupyter, 2017. `http://jupyter.org`.

[38] Yannis Katsis, Yoav Freund, and Yannis Papakonstantinou. Combining databases and signal processing in Plato. In *CIDR*, 2015.

[39] M. Kjelso, M. Gooch, and S. Jones. Design and performance of a main memory hardware data compressor. In *Proceedings of EUROMICRO 96. 22nd Euromicro Conference. Beyond 2000: Hardware and Software Design Strategies*, pages 423–430, 1996.

[40] Donald E Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6(2):163 – 180, 1985.

[41] Interactive Data Lab. Vega: A Visualization Grammar, 2017. `https://vega.github.io/vega`.

[42] Tobin J. Lehman and Michael J. Carey. Query processing in main memory database management systems. *SIGMOD Rec.*, 15(2):239–250, June 1986.

[43] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2122–2131, Dec 2014.

[44] G. Luo, K. Yi, S. Cheng, Z. Li, W. Fan, C. He, and Y. Mu. Piecewise linear approximation of streaming time series data with max-error guarantees. In *2015 IEEE 31st International Conference on Data Engineering*, pages 173–184, 2015.

[45] A. Marascu, P. Pompey, E. Bouillet, M. Wurst, O. Verscheure, M. Grund, and P. Cudre-Mauroux. TRISTAN: Real-time analytics on massive time series using sparse dictionary compression. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 291–300, 2014.

[46] Wes McKinney. pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14(9), 2011.

[47] Sparsh Mittal and Jeffrey S Vetter. A survey of architectural approaches for data compression in cache and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1524–1536, May 2016.

[48] Doron Nakar and Shlomo Weiss. Selective main memory compression by identifying program phase changes. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, WMPI '04, page 96–101, New York, NY, USA, 2004. Association for Computing Machinery.

[49] NumPy. NumPy, 2020. `https://numpy.org/`.

[50] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[51] Oracle. Java, 2020. `https://java.com/`.

[52] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. Weld: Rethinking the interface between data-intensive applications. *CoRR*, abs/1709.06416, 2017.

[53] T. G. Papaioannou, M. Riahi, and K. Aberer. Towards online multi-model approximation of time series. In *2011 IEEE 12th International Conference on Mobile Data Management*, volume 1, pages 33–38, 2011.

[54] Jianzhong Qi, Rui Zhang, Kotagiri Ramamohanarao, Hongzhi Wang, Zeyi Wen, and Dan Wu. Indexable online time series segmentation with error bound guarantee. *World Wide Web*, 18(2):359–401, 2015.

[55] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* McGraw-Hill, third edition, 2000.

[56] S. Ristov and D. Lauc. A system for compacting phonebook database. In *Proceedings of the 25th International Conference on Information Technology Interfaces, 2003. ITI 2003.*, pages 155–159, 2003.

[57] Strahil Ristov. LZ trie and dictionary compression. *Software: Practice and Experience*, 35(5):445–465, 2005.

[58] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Reactive Vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, Jan 2016.

[59] William J Schroeder, Kenneth M Martin, and William E Lorensen. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *Proceedings of the 7th Conference on Visualization '96*, VIS '96, pages 93–100., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[60] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2010.

[61] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, Jan 2002.

[62] Arvind Thiagarajan and Samuel Madden. Querying continuous functions in a database system. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 791–804, New York, NY, USA, 2008. Association for Computing Machinery.

[63] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[64] TriMet. TriMet, 2020. `https://trimet.org/`.

[65] Patrick Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, June 1987.

[66] Sebastiano Vigna. Quasi-succinct indices. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, page 83–92, New York, NY, USA, 2013. Association for Computing Machinery.

[67] Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson. MILC: Inverted list compression in memory. *Proc. VLDB Endow.*, 10(8):853–864, April 2017.

[68] Paul R Wilson, Scott F Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *USENIX Annual Technical Conference, General Track*, pages 101–116, 1999.

[69] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):649–658, Jan 2016.

[70] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, page 258–265, New York, NY, USA, 2000. Association for Computing Machinery.

[71] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 15–28, Berkeley, CA, USA, 2012. USENIX Association.

[72] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[73] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59, 2006.

APPENDIX: DATA-ANALYSIS USE CASE

In this appendix we briefly describe the realistic data-analysis use case that we used to test our system prototype. In addition we present all the individual steps (queries) that we took during the analysis process to reach the final goal. We performed this data analysis on an actual use-case a while ago for a class project using PostgreSQL [28]. First, we will provide a quick overview of the data analysis that we did and talk about the objectives of the analysis and the lessons that we learned. Then, we will show the original data analysis that we did using PostgreSQL as anyone would typically use the system. Then we show the equivalent process in jSQL$_e$. The process involves breaking down the original complex queries into individual operators and keeping all intermediate results in main memory. After that, we show how we were able to perform a similar process (keeping all intermediate results) in three systems, MySQL [18] with in-memory tables, PostgreSQL, and Spark [71]. We do not talk about the performance results of the analysis in this appendix; see Chapter 7 to know more about the performance results.

A.1 DATA-ANALYSIS OVERVIEW

In this section we briefly describe the data analysis and the goals that we set to achieve. In short, the analysis is about figuring out a model that we can use to accurately predict future transit-arrival times using historical data.

There are many apps that we can use to find out the next arrival time of a given bus at a given stop. During certain times of the day, those predictions can be accurate, but not so much during other times. The inaccuracy becomes particularly

problematic when we try to predict arrival times for passenger commuting routes, and even worse when we try to predict arrival times for the distant future such as tomorrow or two days from now. The reason why the predictors from these apps are not good at estimating future arrival times, especially distant future ones, is because they rely on a fixed bus schedule and the real-time geo-position of the buses. The further in the future we go, the less effective the geo-position information becomes and, therefore, the less accurate the predictions are.

The hypothesis that we set to prove or disprove is that traffic, for the most part, has repetitive patterns. If we can capture those patterns, we can use historical data to predict current arrival times (even future ones) with very good accuracy. Examples of repetitive patterns are holidays, weather and seasons, start and end of school, bus-driving behavior, buying groceries, going to and leaving from work, and so on. The idea is to figure out when each pattern occurs and what percentage each pattern contributes to the overall behavior of the traffic flow. The more patterns we capture, the more accurate we can get at predicting the traffic-flow behavior.

The goal is to build a model where we give it a time (present or future), a bus stop, and a route number and it returns the nearest arrival time after the given time. We also want the arrival time to be accurate within ± 3 minutes. The model that we want to build is only for general-traffic-behavior patterns (dining out, going to work, etc). The idea is that once we figure out how to build a model for one pattern, we can build models for other patterns and combine their predictions using different weights to come up with the final prediction.

To achieve our goal, we analyzed six months of transit data from TriMet (Portland, Oregon's public transportation agency) [64]. Similar to machine learning, we split the data into two parts, one part to train the models and the other to test the accuracy of the models. There were three main questions that we wanted to answer:

1. What is the ideal historical period to predict the arrival times for a given day? For example, do we get more accurate predictions if we use the six months of

data prior to the given day or just the three months?

2. What are the right metrics to use to compute arrival times?

3. Should we use data from a given week day to predict arrival times for the same week day? Or, is it more accurate to use data from all weekdays to predict arrival times for a weekday, and use data from weekends to predict arrival times for weekends?

We explain in detail each step of the data analysis that we did in Section A.3.

A.1.1 Lessons Learned

There are a lot of interesting lessons that we learned from this analysis (Section A.3). But the important ones are the following:

- The general-traffic-behavior patterns seems to account for about 89% of all variations. Or to be more specific, just by using the model that we created for the general-traffic-behavior patterns, we were able to make accurate predictions within ± 3 minutes 89% of the time.

- For general-traffic-behavior patterns, we found that data older than 2 months old (from the day whose arrival times to be predicted) makes the models less accurate. Also using less than 2 months of data makes the predictions more accurate, but we get less coverage. That is, we get fewer predictions that fall within ± 3 minutes, but for those that do fall within ± 3 minutes, the percentage increases for the predictions that fall within 0 and ± 1 minute.

- The models we built are good for predicting times that are a week in the future with the same ± 3 minute accuracy. The further in the future we go, the less accurate the predictions start to be come. This observation suggests that the models should be recomputed every week to maintain the level of accuracy.

- We ended up with two models. The first uses each day of the week to predict the same day of the week. For example, we use Mondays to predict arrival times on Mondays. The second model uses all weekdays to predict any arrival time during a weekday, uses Saturdays to predict arrival times on Saturdays, and uses Sundays to predict arrival times on Sundays. From the testing that we did, overall, the first model seems to be more accurate than the second model.

For the rest of this Appendix, we show the actual analysis (the queries) that we originally did in addition to the simulated analysis that we did on the other systems.

A.2 DATA SCHEMA

As we mentioned earlier, the data that we used is TriMet's [64] daily public-transit data. The following is the schema of the data. The original data set has more columns than we list here, but the columns that we list her are the only relevant ones that we used in the analysis.

```
CREATE TABLE stop_event (
  SERVICE_DATE char varying(20),
  LEAVE_TIME integer,
  ROUTE_NUMBER integer,
  STOP_TIME integer,
  ARRIVE_TIME integer,
  LOCATION_ID integer,
  SCHEDULE_STATUS integer
);
```

The field `SERVICE_DATE` is the calendar date on which the data was collected. The field `LEAVE_TIME` is the time of the day at which the bus or train left the bus or train stop. The field `ROUTE_NUMBER` is the bus or train route number. The field `STOP_TIME` is the time of the day at which the bus or train is scheduled to arrive at the bus or train stop. The field `ARRIVE_TIME` is the time of the day at which the bus or train arrived

at the bus or train stop. The field `LOCATION_ID` is the bus or train stop id. The field `SCHEDULE_STATUS` is the type of schedule (e.g., weekday, Saturday, Sunday, or holiday schedule) that was used for that day. The values in the fields `LEAVE_TIME`, `STOP_TIME`, and `ARRIVE_TIME` are expressed in number of seconds since 12am of a given day.

A.3 ORIGINAL ANALYSIS

The following is the original analysis as it was done using PostgreSQL [28]. If the reader is interested in the models that ended up working well, the models are STMT 19 and STMT 20. However STMT 19 seems to yield more accurate results.

STMT 1: The first statement is trying to understand the data better. So we pick a certain known stop for a given route and compare the result to what we except from our experience.

```
1 -- STMT: 1
2 SELECT * FROM stop_event
3 WHERE
4   service_date = '2018-12-10' AND
5   route_number = 58 AND
6   LOCATION_ID = 910
7 ORDER BY arrive_time;
```

STMT 2: We continue to try to understand the data. Here we pick a known stop where we expect to see multiple routes and compare the results to what we except.

```
1 -- STMT: 2
2 SELECT DISTINCT route_number
3 FROM stop_event
4 WHERE service_date = '2018-12-10' AND LOCATION_ID = 9821;
```

STMT 3: Here we want to see which assumptions that we have about the data are true and which are not. In this next statement we are checking to see if there is only one observation for each route at a given stop at a given day at a given schedule time.

```sql
-- STMT: 3
SELECT
  t1.SERVICE_DATE,
  t1.ROUTE_NUMBER,
  t1.LOCATION_ID,
  t1.STOP_TIME,
  count(*)
FROM
  stop_event t1
GROUP BY
  t1.SERVICE_DATE,
  t1.ROUTE_NUMBER,
  t1.LOCATION_ID,
  t1.STOP_TIME
HAVING
  count(*) > 1;
```

STMT 4: Continuing to understand the data, we are trying to figure out how to interpret the values in the `STOP_TIME` column by picking a certain stop time and comparing it to the know bus schedule.

```sql
-- STMT: 4
SELECT * FROM  stop_event t1
WHERE
  t1.SERVICE_DATE = '2018-12-02' AND
  t1.ROUTE_NUMBER = 58 AND
  t1.LOCATION_ID =  12790 AND
  t1.STOP_TIME = 38280;
```

STMT 5: Continuing to examine our assumptions. Here we check to see if a known stop serves multiple routes.

```sql
-- STMT: 5
SELECT DISTINCT route_number
FROM stop_event
```

```
WHERE
  service_date = '2018-12-10' AND
  LOCATION_ID = 9818;
```

STMT 6: Here we start with quick statistics just to get a sense of the range of delays that we see in the data. So the statement builds a histogram for each day of the week for each route for each stop for each schedule time for each delay value within one-minute increments.

```
-- STMT: 6
-- Creating a historgram
DROP TABLE stop_event_histogram;
CREATE TABLE stop_event_histogram AS
SELECT
  -- 0: sun, 1:mon, ... , 6: sat
  extract(dow FROM SERVICE_DATE) day_of_week,
  ROUTE_NUMBER,
  LOCATION_ID,
  STOP_TIME,
  -- The delay time in seconds. The time rounded down to a minute.
  TRUNC((ARRIVE_TIME - STOP_TIME) / 60)::int * 60 AS delay,
  count(*) num_of_observations
FROM
  stop_event
GROUP BY
  day_of_week,
  ROUTE_NUMBER,
  LOCATION_ID,
  STOP_TIME,
  delay;
```

STMT 7: The next statement is the first attempt to create a model. For each day of the week for each route for each stop for each schedule time, compute the average delay for three months of data excluding the holiday period (outliers).

```sql
-- STMT: 7
-- MODEL 1: Creating avg delay per week day.
DROP TABLE stop_event_avg_delay;
CREATE TABLE stop_event_avg_delay AS
SELECT
  -- 0: sun, 1:mon, ... , 6: sat
  extract(dow FROM SERVICE_DATE) day_of_week,
  ROUTE_NUMBER,
  LOCATION_ID,
  STOP_TIME,
  TRUNC(avg(ARRIVE_TIME - STOP_TIME))::int AS avg_delay,
  count(*) num_of_observations
FROM
  stop_event
WHERE
  (
    SERVICE_DATE >= '2018-11-01' AND SERVICE_DATE < '2018-12-15' OR
    SERVICE_DATE >= '2019-01-10' AND SERVICE_DATE < '2019-02-01'
  )
GROUP BY
  day_of_week,
  ROUTE_NUMBER,
  LOCATION_ID,
  STOP_TIME;
```

STMT 8: The previous attempt (STMT 7) was not successful because there were many outliers that made the predictions way off with respect to the actual arrival time. So in this statement we clean up the outliers first before we compute the average. The first step is to figure out which time is the closest to the scheduled time, arrive time or leave time. Then we compute the delay based on the closest time. We also remove route 0 because it is for maintenance. Next, for each service date for each route for each stop for each schedule time, we pick the observation with the shortest

delay. The final step in the cleaning process is to pick the observations that are only within one standard deviation from the average. Then we compute the average on the remaining observations.

```
1  -- STMT: 8
2  -- MODEL 1: Creating average arrival times and leave times per week day
       .
3  DROP TABLE stop_event_avg_delay;
4  CREATE TABLE stop_event_avg_delay AS
5  WITH base_data AS (
6    SELECT
7      SERVICE_DATE,
8      -- 0: sun, 1:mon, ... , 6: sat
9      extract(dow FROM SERVICE_DATE) day_of_week,
10     ROUTE_NUMBER,
11     LOCATION_ID,
12     STOP_TIME,
13     CASE
14       WHEN abs(ARRIVE_TIME - STOP_TIME) <= abs(LEAVE_TIME - STOP_TIME)
         THEN
15         ARRIVE_TIME - STOP_TIME
16       ELSE
17         LEAVE_TIME - STOP_TIME
18     END AS delay
19   FROM
20     stop_event
21   WHERE
22     (
23       SERVICE_DATE >= '2018-12-01' AND SERVICE_DATE < '2018-12-15' OR
24       SERVICE_DATE >= '2019-01-10' AND SERVICE_DATE < '2019-02-01'
25     ) AND
26     ROUTE_NUMBER <> 0
27 ), base_data_with_min_delay AS (
```

```
  SELECT
    t1.*,
    min(abs(delay)) OVER(PARTITION BY SERVICE_DATE, ROUTE_NUMBER,
     LOCATION_ID, STOP_TIME) AS abs_min_delay
  FROM
    base_data AS t1
), cleaned_base_data AS (
  SELECT
    SERVICE_DATE,
    day_of_week,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    min(delay) AS delay
  FROM
    base_data_with_min_delay
  WHERE
    abs(delay) = abs_min_delay
  GROUP BY
    SERVICE_DATE,
    day_of_week,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
), base_model AS (
  SELECT
    day_of_week,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    stddev(delay) AS std_delay,
    avg(delay) AS avg_delay
```

```
  FROM
    cleaned_base_data
  GROUP BY
    day_of_week,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
)
SELECT
  t2.day_of_week,
  t2.ROUTE_NUMBER,
  t2.LOCATION_ID,
  t2.STOP_TIME,
  TRUNC(COALESCE(avg(t1.delay), t2.avg_delay))::int AS avg_delay
FROM
  base_model t2
  LEFT JOIN cleaned_base_data t1
    ON  t1.day_of_week = t2.day_of_week AND
      t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
      t1.LOCATION_ID = t2.LOCATION_ID AND
      t1.STOP_TIME = t2.STOP_TIME AND
      abs(t1.delay) <= abs(t2.avg_delay) + t2.std_delay
GROUP BY
  t2.day_of_week,
  t2.ROUTE_NUMBER,
  t2.LOCATION_ID,
  t2.STOP_TIME,
  t2.avg_delay;
```

STMT 9: In this statement we check to see if there is a difference in the average delay between the days of the week in the model that we built from STMT 8. Note that we only show the comparison between Tuesday and Wednesday here.

```
-- STMT: 9
```

```
-- compare averages of days of the week.
SELECT
  t1.ROUTE_NUMBER,
  t1.LOCATION_ID,
  t1.STOP_TIME,
  TRUNC(t1.avg_delay / 60)::int as dow1_delay,
  TRUNC(t2.avg_delay / 60)::int as dow2_delay
FROM
  stop_event_avg_delay t1
  JOIN stop_event_avg_delay t2
    ON
      t1.route_number = t2.route_number AND
      t1.location_id = t2.location_id AND
      t1.stop_time = t2.stop_time
WHERE
  t1.day_of_week = 2 AND
  t2.day_of_week = 3 AND
  t1.route_number = 78
ORDER BY
  LOCATION_ID, STOP_TIME
```

STMT 10: From STMT 9, it seemed that the delay difference is not significant among weekdays but it is significant compared to weekends. So in this statement we build a second model similar to STMT 8 but instead of having a prediction for each day of the week, we have one for weekdays, one for Saturdays, and one for Sundays. The predictions for weekdays are computed based on all the observations from all weekdays.

```
-- STMT: 10
-- MODEL 2:  Creating average arrival times and leave times for
    weekdays and another for sat and sun.
DROP TABLE stop_event_avg_delay_dow_class;
CREATE TABLE stop_event_avg_delay_dow_class AS
```

```sql
WITH base_data AS (
  SELECT
    SERVICE_DATE,
    -- D: weekday, S:saturday, U: sunday
    CASE
      WHEN extract(dow FROM SERVICE_DATE) IN (1,2,3,4,5) THEN 'D'
      WHEN extract(dow FROM SERVICE_DATE) = 0 THEN 'U'
      ELSE 'S'
    END AS dow_class,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    CASE
      WHEN abs(ARRIVE_TIME - STOP_TIME) <= abs(LEAVE_TIME - STOP_TIME)
     THEN
        ARRIVE_TIME - STOP_TIME
      ELSE
        LEAVE_TIME - STOP_TIME
    END AS delay
  FROM
    stop_event
  WHERE
    (
      SERVICE_DATE >= '2018-12-01' AND SERVICE_DATE < '2018-12-15' OR
      SERVICE_DATE >= '2019-01-10' AND SERVICE_DATE < '2019-02-01'
    ) AND
    ROUTE_NUMBER <> 0
), base_data_with_min_delay AS (
  SELECT
    t1.*,
    min(abs(delay)) OVER(PARTITION BY SERVICE_DATE, ROUTE_NUMBER,
     LOCATION_ID, STOP_TIME) AS abs_min_delay
```

```
  FROM
    base_data AS t1
), cleaned_base_data AS (
  SELECT
    SERVICE_DATE,
    dow_class,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    min(delay) AS delay
  FROM
    base_data_with_min_delay
  WHERE
    abs(delay) = abs_min_delay
  GROUP BY
    SERVICE_DATE,
    dow_class,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
), base_model AS (
  SELECT
    dow_class,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    stddev(delay) AS std_delay,
    avg(delay) AS avg_delay
  FROM
    cleaned_base_data
  GROUP BY
    dow_class,
```

```
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
)
SELECT
  t2.dow_class,
  t2.ROUTE_NUMBER,
  t2.LOCATION_ID,
  t2.STOP_TIME,
  TRUNC(COALESCE(avg(t1.delay), t2.avg_delay))::int AS avg_delay
FROM
  base_model t2
  LEFT JOIN cleaned_base_data t1
    ON  t1.dow_class = t2.dow_class AND
      t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
      t1.LOCATION_ID = t2.LOCATION_ID AND
      t1.STOP_TIME = t2.STOP_TIME AND
      abs(t1.delay) <= abs(t2.avg_delay) + t2.std_delay
GROUP BY
  t2.dow_class,
  t2.ROUTE_NUMBER,
  t2.LOCATION_ID,
  t2.STOP_TIME,
  t2.avg_delay;
```

STMT 11: We also wanted to try something similar to what we did in STMT 10 but for STMT 7 instead of STMT 8, just to see if grouping weekdays makes a difference. We also do not exclude the holiday period in this statement.

```
-- STMT: 11
-- MODEL 2: Creating an avg delay for weekdays and another for sat and
    sun.
DROP TABLE stop_event_avg_delay_dow_class;
CREATE TABLE stop_event_avg_delay_dow_class AS
```

```
SELECT
  -- D: weekday, S:saturday, U: sunday
  CASE
    WHEN extract(dow FROM SERVICE_DATE) IN (1,2,3,4,5) THEN 'D'
    WHEN extract(dow FROM SERVICE_DATE) = 0 THEN 'U'
    ELSE 'S'
  END AS dow_class,
  ROUTE_NUMBER,
  LOCATION_ID,
  STOP_TIME,
  TRUNC(avg(ARRIVE_TIME - STOP_TIME))::int AS avg_delay,
  count(*) num_of_observations
FROM
  stop_event
WHERE
  SERVICE_DATE >= '2018-11-01' AND SERVICE_DATE < '2019-02-01'
GROUP BY
  dow_class,
  ROUTE_NUMBER,
  LOCATION_ID,
  STOP_TIME;
```

STMT 12: In this statement we compare both models 1 (STMT 8) and 2 (both STMT 10 and 11). Here we picked a certain week day (the day on which we ran this query) and compared the real-time arrival time for two routes for a certain stop to see which model was the closest.

```
-- STMT: 12
-- compare averages of day of week vs. weekdays and weekends.
SELECT
  t1.ROUTE_NUMBER,
  t1.LOCATION_ID,
  t1.STOP_TIME as stop_t_sec,
```

```
  t1.STOP_TIME * interval '1 sec' AS stop_time,
  TRUNC(t1.avg_delay / 60)::int as dow1_delay,
  TRUNC(t2.avg_delay / 60)::int as dow2_delay
FROM
  stop_event_avg_delay t1
  JOIN stop_event_avg_delay_dow_class t2
    ON
      t1.route_number = t2.route_number AND
      t1.location_id = t2.location_id AND
      t1.stop_time = t2.stop_time
WHERE
  t1.day_of_week = 5 AND
  t2.dow_class = 'D' AND
  t1.route_number in (76,78) AND
  t1.location_id = 2285
ORDER BY
  LOCATION_ID, STOP_TIME
```

STMT 13: The comparison we did in STMT 12 showed promising results, but we only tested one stop. So we need to check the predictions for all stops for all routes for all schedule times. So the first step is to create a baseline to which we are going to compare our model predictions. The baseline is going to be delays with respect to the schedule. Here we simply gather statistics for each delay value, rounded to a minute. We also use a period of time that was not used to train the model.

```
-- STMT: 13
-- Create a baseline measure for all hours.
WITH diffs AS (
  SELECT
    TRUNC((ARRIVE_TIME - STOP_TIME) / 60)::int AS delay_diff,
    count(*) AS observations
  FROM
    stop_event
```

```
  WHERE
    SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01'
  GROUP BY
    delay_diff
)
SELECT
  CASE
    WHEN abs(delay_diff) > 5 THEN 'others'
    ELSE delay_diff::text
  END AS delay_diffS,
  SUM(observations) AS observations
FROM
  diffs
GROUP BY
  delay_diffS
ORDER BY
  delay_diffs;
```

STMT 14: This statement is similar to STMT 13, but we just want a baseline for rush hours because those hours are when the longest delays occur.

```
-- STMT: 14
-- Create a baseline measure for rush hours.
SELECT
  TRUNC((ARRIVE_TIME - STOP_TIME) / 60)::int AS delay_diff,
  count(*) AS observations
FROM
  stop_event
WHERE
  SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01' AND
  (
    -- 23160 = 06:26:00, 31140 = 08:39:00
    STOP_TIME >= 23160 AND STOP_TIME <= 31140 OR
    -- 57600 = 16:00:00, 66780 = 18:33:00
```

```
    STOP_TIME >= 57600 AND STOP_TIME <= 66780
  )
GROUP BY
  delay_diff
ORDER BY
  observations desc;
```

STMT 15: Here we compare the actual arrival times for a month (whose data was not used to train the model) to the predicted arrival times using Model 1 (STMT 8). Then we gather statistics on how far off our predictions are from the actual arrival times.

```
-- STMT: 15
-- Compare predictions from model 1 to actual arrival times during one
    month
WITH feb_data AS (
  SELECT
    extract(dow FROM SERVICE_DATE) AS day_of_week,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    TRUNC((CASE
      WHEN abs(ARRIVE_TIME - STOP_TIME) <= abs(LEAVE_TIME - STOP_TIME)
     THEN
        ARRIVE_TIME - STOP_TIME
      ELSE
        LEAVE_TIME - STOP_TIME
    END) / 60)::int AS actual_delay_in_min
  FROM
    stop_event
  WHERE
    SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01' AND
    ROUTE_NUMBER <> 0 AND
```

```
    schedule_status <> 6
), diffs AS (
  SELECT
    TRUNC(t2.avg_delay / 60)::int - t1.actual_delay_in_min AS
     delay_diff,
    count(*) AS opservations
  FROM
    feb_data AS t1
    JOIN stop_event_avg_delay AS t2
      ON  t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
        t1.LOCATION_ID = t2.LOCATION_ID AND
        t1.STOP_TIME = t2.STOP_TIME AND
        t1.day_of_week = t2.day_of_week
  GROUP BY
    delay_diff
)
SELECT
  CASE
    WHEN abs(delay_diff) > 3 THEN 'others'
    ELSE delay_diff::text
  END AS delay_diffS,
  SUM(opservations) AS opservations
FROM
  diffs
GROUP BY
  delay_diffS
ORDER BY
  delay_diffs;
```

STMT 16: Here we do something similar to STMT 15 but for rush hours only.

```
-- STMT: 16
-- Compare predictions from model 1 to actual arrival times during one
    month for rush hours
```

```sql
WITH feb_data AS (
  SELECT
    extract(dow FROM SERVICE_DATE) AS day_of_week,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    TRUNC((ARRIVE_TIME - STOP_TIME) / 60)::int AS actual_delay_in_min
  FROM
    stop_event
  WHERE
    SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01' AND
    (
      -- 23160 = 06:26:00, 31140 = 08:39:00
      STOP_TIME >= 23160 AND STOP_TIME <= 31140 OR
      -- 57600 = 16:00:00, 66780 = 18:33:00
      STOP_TIME >= 57600 AND STOP_TIME <= 66780
    )
)
SELECT
  TRUNC(t2.avg_delay / 60)::int - t1.actual_delay_in_min AS delay_diff,
  count(*) AS observations
FROM
  feb_data AS t1
  JOIN stop_event_avg_delay AS t2
    ON  t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
      t1.LOCATION_ID = t2.LOCATION_ID AND
      t1.STOP_TIME = t2.STOP_TIME AND
      t1.day_of_week = t2.day_of_week
GROUP BY
  delay_diff
ORDER BY
  observations desc;
```

STMT 17: In this statement we repeat the analysis in STMT 15 but this time we use Model 2 (STMT 10).

```
1  -- STMT: 17
2  -- Compare predictions from model 2 to actual arrival times during one
       month
3  WITH feb_data AS (
4    SELECT
5      CASE
6       WHEN extract(dow FROM SERVICE_DATE) IN (1,2,3,4,5) THEN 'D'
7       WHEN extract(dow FROM SERVICE_DATE) = 0 THEN 'U'
8       ELSE 'S'
9     END AS dow_class,
10    ROUTE_NUMBER,
11    LOCATION_ID,
12    STOP_TIME,
13    TRUNC((CASE
14      WHEN abs(ARRIVE_TIME - STOP_TIME) <= abs(LEAVE_TIME - STOP_TIME)
        THEN
15        ARRIVE_TIME - STOP_TIME
16      ELSE
17        LEAVE_TIME - STOP_TIME
18    END) / 60)::int AS actual_delay_in_min
19   FROM
20    stop_event
21   WHERE
22    SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01' AND
23    ROUTE_NUMBER <> 0 AND
24    schedule_status <> 6
25 ), diffs AS (
26   SELECT
27    TRUNC(t2.avg_delay / 60)::int - t1.actual_delay_in_min AS
       delay_diff,
```

```
    count(*) AS opservations
  FROM
    feb_data AS t1
    JOIN stop_event_avg_delay_dow_class AS t2
      ON  t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
        t1.LOCATION_ID = t2.LOCATION_ID AND
        t1.STOP_TIME = t2.STOP_TIME AND
        t1.dow_class = t2.dow_class
  GROUP BY
    delay_diff
)
SELECT
  CASE
    WHEN abs(delay_diff) > 3 THEN 'others'
    ELSE delay_diff::text
  END AS delay_diffS,
  SUM(opservations) AS opservations
FROM
  diffs
GROUP BY
  delay_diffS
ORDER BY
  delay_diffs;
```

STMT 18: This is similar to STMT 16 but using Model 2 (STMT 10) instead.

```
-- STMT: 18
-- Compare predictions from model 2 to actual arrival times during one
    month for rush hours
WITH feb_data AS (
  SELECT
    CASE
      WHEN extract(dow FROM SERVICE_DATE) IN (1,2,3,4,5) THEN 'D'
      WHEN extract(dow FROM SERVICE_DATE) = 0 THEN 'U'
```

```
        ELSE 'S'
    END AS dow_class,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    TRUNC((ARRIVE_TIME - STOP_TIME) / 60)::int AS actual_delay_in_min
  FROM
    stop_event
  WHERE
    SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01' AND
    (
      -- 23160 = 06:26:00, 31140 = 08:39:00
      STOP_TIME >= 23160 AND STOP_TIME <= 31140 OR
      -- 57600 = 16:00:00, 66780 = 18:33:00
      STOP_TIME >= 57600 AND STOP_TIME <= 66780
    )
)
SELECT
  TRUNC(t2.avg_delay / 60)::int - t1.actual_delay_in_min AS delay_diff,
  count(*) AS observations
FROM
  feb_data AS t1
  JOIN stop_event_avg_delay_dow_class AS t2
    ON  t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
      t1.LOCATION_ID = t2.LOCATION_ID AND
      t1.STOP_TIME = t2.STOP_TIME AND
      t1.dow_class = t2.dow_class
GROUP BY
  delay_diff
ORDER BY
  observations desc;
```

STMT 19: The first approach of building the model (STMT 8 and 10) was not that

much better from the baseline (using the schedule to predict arrival times). Mostly the reason was because of how we chose to eliminate duplicates. In this statement we use a different approach (Model 1 v2) to remove duplicates, which turned out to have much better predictions. First we eliminate duplicates by creating a new observation for each group of duplicate values such that the arrive time is the minimum arrive time and the leave time is the maximum leave time in each group. Then we compute the standard deviation and the average for both the arrive time and leave time over the newly-created observations. Then we compute the predicted arrive time and leave time by computing the average for each, but only for the values that are within one standard deviation from the previously computed average.

```
1  -- STMT: 19
2  -- MODEL 1: Create the model for every day of the week.
3  DROP TABLE stop_event_avg_delay;
4  CREATE TABLE stop_event_avg_delay AS
5  WITH base_unique_data AS (
6    SELECT
7      SERVICE_DATE,
8      ROUTE_NUMBER,
9      LOCATION_ID,
10     STOP_TIME,
11     min(ARRIVE_TIME) AS ARRIVE_TIME,
12     max(LEAVE_TIME) AS LEAVE_TIME,
13     -- 0: sun, 1:mon, ... , 6: sat
14     extract(dow FROM SERVICE_DATE) day_of_week
15   FROM
16     stop_event
17   WHERE
18     (
19       SERVICE_DATE >= '2018-12-01' AND SERVICE_DATE < '2018-12-15' OR
20       SERVICE_DATE >= '2019-01-10' AND SERVICE_DATE < '2019-02-01'
21     ) AND
```

```
    ROUTE_NUMBER <> 0
  GROUP BY
    SERVICE_DATE,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
), base_model AS (
  SELECT
    day_of_week,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    stddev(ARRIVE_TIME) AS std_arrive_time,
    avg(ARRIVE_TIME) AS avg_arrive_time,
    stddev(LEAVE_TIME) AS std_leave_time,
    avg(LEAVE_TIME) AS avg_leave_time
  FROM
    base_unique_data
  GROUP BY
    day_of_week,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
)
SELECT
  t2.day_of_week,
  t2.ROUTE_NUMBER,
  t2.LOCATION_ID,
  t2.STOP_TIME,
  TRUNC(COALESCE(
    avg(t1.ARRIVE_TIME) FILTER(WHERE abs(t1.ARRIVE_TIME) <= abs(t2.
    avg_arrive_time) + t2.std_arrive_time),
```

```
    t2.avg_arrive_time
  ))::int AS arrive_time,
  TRUNC(COALESCE(
    avg(t1.LEAVE_TIME) FILTER(WHERE abs(t1.LEAVE_TIME) <= abs(t2.
     avg_leave_time) + t2.std_leave_time),
    t2.avg_leave_time
  ))::int AS leave_time
FROM
  base_model t2
  LEFT JOIN base_unique_data t1
    ON  t1.day_of_week = t2.day_of_week AND
      t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
      t1.LOCATION_ID = t2.LOCATION_ID AND
      t1.STOP_TIME = t2.STOP_TIME
GROUP BY
  t2.day_of_week,
  t2.ROUTE_NUMBER,
  t2.LOCATION_ID,
  t2.STOP_TIME,
  t2.avg_arrive_time,
  t2.avg_leave_time;
```

STMT 20: This statement is similar to STMT 19 but instead of having a prediction for each day of the week, we have one prediction for weekdays, one for Saturdays, and one for Sundays. This statement is basically Model 2 v2.

```
-- STMT: 20
-- MODEL 2: Create the model for weekdays, Saturdays, and for Sundays
DROP TABLE stop_event_avg_delay_dow_class;
CREATE TABLE stop_event_avg_delay_dow_class AS
WITH base_unique_data AS (
  SELECT
    SERVICE_DATE,
    ROUTE_NUMBER,
```

```
    LOCATION_ID,
    STOP_TIME,
    min(ARRIVE_TIME) AS ARRIVE_TIME,
    max(LEAVE_TIME) AS LEAVE_TIME,
    -- D: weekday, S:saturday, U: sunday
    CASE
      WHEN extract(dow FROM SERVICE_DATE) IN (1,2,3,4,5) THEN 'D'
      WHEN extract(dow FROM SERVICE_DATE) = 0 THEN 'U'
      ELSE 'S'
    END AS dow_class
  FROM
    stop_event
  WHERE
    (
      SERVICE_DATE >= '2018-12-01' AND SERVICE_DATE < '2018-12-15' OR
      SERVICE_DATE >= '2019-01-10' AND SERVICE_DATE < '2019-02-01'
    ) AND
    ROUTE_NUMBER <> 0
  GROUP BY
    SERVICE_DATE,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
), base_model AS (
  SELECT
    dow_class,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    stddev(ARRIVE_TIME) AS std_arrive_time,
    avg(ARRIVE_TIME) AS avg_arrive_time,
    stddev(LEAVE_TIME) AS std_leave_time,
```

```
    avg(LEAVE_TIME) AS avg_leave_time
  FROM
    base_unique_data
  GROUP BY
    dow_class,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
)
SELECT
  t2.dow_class,
  t2.ROUTE_NUMBER,
  t2.LOCATION_ID,
  t2.STOP_TIME,
  TRUNC(COALESCE(
    avg(t1.ARRIVE_TIME) FILTER(WHERE abs(t1.ARRIVE_TIME) <= abs(t2.
    avg_arrive_time) + t2.std_arrive_time),
    t2.avg_arrive_time
  ))::int AS arrive_time,
  TRUNC(COALESCE(
    avg(t1.LEAVE_TIME) FILTER(WHERE abs(t1.LEAVE_TIME) <= abs(t2.
    avg_leave_time) + t2.std_leave_time),
    t2.avg_leave_time
  ))::int AS leave_time
FROM
  base_model t2
  LEFT JOIN base_unique_data t1
    ON  t1.dow_class = t2.dow_class AND
      t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
      t1.LOCATION_ID = t2.LOCATION_ID AND
      t1.STOP_TIME = t2.STOP_TIME
GROUP BY
```

```
    t2.dow_class,
    t2.ROUTE_NUMBER,
    t2.LOCATION_ID,
    t2.STOP_TIME,
    t2.avg_arrive_time,
    t2.avg_leave_time;
```

STMT 21: Here we create another baseline, but this time we also eliminate duplicates by creating a new observation for each group of duplicate values such that the arrive time is the minimum arrive time and the leave time is the maximum leave time in each group.

```
-- STMT: 21
-- Create a baseline measure for all hours.
WITH feb_data AS (
  SELECT
    SERVICE_DATE,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    min(ARRIVE_TIME) AS arrive_time,
    max(LEAVE_TIME) AS leave_time
  FROM
    stop_event
  WHERE
    SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01' AND
    ROUTE_NUMBER <> 0
  GROUP BY
    SERVICE_DATE,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
), diffs AS (
```

```
  SELECT
    TRUNC(CASE
      WHEN abs(arrive_time - STOP_TIME) <= abs(leave_time - 30 -
     STOP_TIME) THEN
        arrive_time - STOP_TIME
      ELSE
        leave_time - 30 - STOP_TIME
    END  / 60)::int AS prediction_diff
  FROM
    feb_data
)
SELECT
  CASE
    WHEN abs(prediction_diff) > 3 THEN 'others'
    ELSE prediction_diff::text
  END AS prediction_diffs,
  count(*) AS opservations
FROM
  diffs
GROUP BY
  prediction_diffs
ORDER BY
  prediction_diffs;
```

STMT 22: Here we create another baseline similar to STMT 21 but only for rush hours.

```
-- STMT: 22
-- Create a baseline measure for rush hours.
WITH feb_data AS (
  SELECT
    SERVICE_DATE,
    ROUTE_NUMBER,
    LOCATION_ID,
```

```
    STOP_TIME,
    min(ARRIVE_TIME) AS arrive_time,
    max(LEAVE_TIME) AS leave_time,
    -- 0: sun, 1:mon, ... , 6: sat
    extract(dow FROM SERVICE_DATE) day_of_week
  FROM
    stop_event
  WHERE
    SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01' AND
    ROUTE_NUMBER <> 0 AND
    (
      -- 23160 = 06:26:00, 31140 = 08:39:00
      STOP_TIME >= 23160 AND STOP_TIME <= 31140 OR
      -- 57600 = 16:00:00, 66780 = 18:33:00
      STOP_TIME >= 57600 AND STOP_TIME <= 66780
    )
  GROUP BY
    SERVICE_DATE,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
), diffs AS (
  SELECT
    TRUNC(CASE
      WHEN abs(arrive_time - STOP_TIME) <= abs(leave_time - 30 -
    STOP_TIME) THEN
        arrive_time - STOP_TIME
      ELSE
        leave_time - 30 - STOP_TIME
    END  / 60)::int AS prediction_diff
  FROM
    feb_data
```

```
)
SELECT
  CASE
    WHEN abs(prediction_diff) > 3 THEN 'others'
    ELSE prediction_diff::text
  END AS prediction_diffs,
  count(*) AS opservations
FROM
  diffs
GROUP BY
  prediction_diffs
ORDER BY
  prediction_diffs;
```

STMT 23: In this statement we compare Model 1 v2 (STMT 19) to the actual arrival times.

```
-- STMT: 23
-- Compare predictions from model 1 to actual arrival times during one
    month for all hours
WITH feb_data AS (
  SELECT
    SERVICE_DATE,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    min(ARRIVE_TIME) AS arrive_time,
    max(LEAVE_TIME) AS leave_time,
    -- 0: sun, 1:mon, ... , 6: sat
    extract(dow FROM SERVICE_DATE) day_of_week
  FROM
    stop_event
  WHERE
    SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01' AND
```

```
    ROUTE_NUMBER <> 0
  GROUP BY
    SERVICE_DATE,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
), diffs AS (
  SELECT
    TRUNC((t2.arrive_time - t1.arrive_time) / 60)::int AS
     prediction_diff,
    count(*) AS opservations
  FROM
    feb_data AS t1
    JOIN stop_event_avg_delay AS t2
      ON  t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
        t1.LOCATION_ID = t2.LOCATION_ID AND
        t1.STOP_TIME = t2.STOP_TIME AND
        t1.day_of_week = t2.day_of_week
  GROUP BY
    prediction_diff
)
SELECT
  CASE
    WHEN abs(prediction_diff) > 3 THEN 'others'
    ELSE prediction_diff::text
  END AS prediction_diffs,
  SUM(opservations) AS opservations
FROM
  diffs
GROUP BY
  prediction_diffs
ORDER BY
```

```
48  prediction_diffs;
```

STMT 24: Here we compare Model 1 v2 (STMT 19) to the actual arrival times but only for rush hours.

```
1  -- STMT: 24
2  -- Compare predictions from model 1 to actual arrival times during one
       month for rush hours
3  WITH feb_data AS (
4    SELECT
5      SERVICE_DATE,
6      ROUTE_NUMBER,
7      LOCATION_ID,
8      STOP_TIME,
9      min(ARRIVE_TIME) AS arrive_time,
10     max(LEAVE_TIME) AS leave_time,
11     -- 0: sun, 1:mon, ... , 6: sat
12     extract(dow FROM SERVICE_DATE) day_of_week
13   FROM
14     stop_event
15   WHERE
16     SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01' AND
17     ROUTE_NUMBER <> 0 AND
18     (
19       -- 23160 = 06:26:00, 31140 = 08:39:00
20       STOP_TIME >= 23160 AND STOP_TIME <= 31140 OR
21       -- 57600 = 16:00:00, 66780 = 18:33:00
22       STOP_TIME >= 57600 AND STOP_TIME <= 66780
23     )
24   GROUP BY
25     SERVICE_DATE,
26     ROUTE_NUMBER,
27     LOCATION_ID,
28     STOP_TIME
```

```
), diffs AS (
  SELECT
    TRUNC((t2.arrive_time - t1.arrive_time) / 60)::int AS
    prediction_diff,
    count(*) AS opservations
  FROM
    feb_data AS t1
    JOIN stop_event_avg_delay AS t2
      ON  t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
        t1.LOCATION_ID = t2.LOCATION_ID AND
        t1.STOP_TIME = t2.STOP_TIME AND
        t1.day_of_week = t2.day_of_week
  GROUP BY
    prediction_diff
)
SELECT
  CASE
    WHEN abs(prediction_diff) > 3 THEN 'others'
    ELSE prediction_diff::text
  END AS prediction_diffs,
  SUM(opservations) AS opservations
FROM
  diffs
GROUP BY
  prediction_diffs
ORDER BY
  prediction_diffs;
```

STMT 25: In this statement we compare Model 2 v2 (STMT 20) to the actual arrival times.

```
-- STMT: 25
-- Compare predictions from model 2 to actual arrival times during one
    month for all hours
```

```
WITH feb_data AS (
  SELECT
    SERVICE_DATE,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME,
    min(ARRIVE_TIME) AS arrive_time,
    max(LEAVE_TIME) AS leave_time,
    -- D: weekday, S:saturday, U: sunday
    CASE
      WHEN extract(dow FROM SERVICE_DATE) IN (1,2,3,4,5) THEN 'D'
      WHEN extract(dow FROM SERVICE_DATE) = 0 THEN 'U'
      ELSE 'S'
    END AS dow_class
  FROM
    stop_event
  WHERE
    SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01' AND
    ROUTE_NUMBER <> 0
  GROUP BY
    SERVICE_DATE,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
), diffs AS (
  SELECT
    TRUNC((t2.arrive_time - t1.arrive_time) / 60)::int AS
    prediction_diff,
    count(*) AS opservations
  FROM
    feb_data AS t1
    JOIN stop_event_avg_delay_dow_class AS t2
```

```
        ON  t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
          t1.LOCATION_ID = t2.LOCATION_ID AND
          t1.STOP_TIME = t2.STOP_TIME AND
          t1.dow_class = t2.dow_class
    GROUP BY
      prediction_diff
)
SELECT
    CASE
      WHEN abs(prediction_diff) > 3 THEN 'others'
      ELSE prediction_diff::text
    END AS prediction_diffs,
    SUM(opservations) AS opservations
FROM
    diffs
GROUP BY
    prediction_diffs
ORDER BY
    prediction_diffs;
```

STMT 26: Here we compare Model 2 v2 (STMT 20) to the actual arrival times but only for rush hours.

```
-- STMT: 26
-- Compare predictions from model 2 to actual arrival times during one
    month for rush hours
WITH feb_data AS (
    SELECT
      SERVICE_DATE,
      ROUTE_NUMBER,
      LOCATION_ID,
      STOP_TIME,
      min(ARRIVE_TIME) AS arrive_time,
      max(LEAVE_TIME) AS leave_time,
```

```
    -- D: weekday, S:saturday, U: sunday
    CASE
      WHEN extract(dow FROM SERVICE_DATE) IN (1,2,3,4,5) THEN 'D'
      WHEN extract(dow FROM SERVICE_DATE) = 0 THEN 'U'
      ELSE 'S'
    END AS dow_class
  FROM
    stop_event
  WHERE
    SERVICE_DATE >= '2019-02-01' AND SERVICE_DATE < '2019-03-01' AND
    ROUTE_NUMBER <> 0 AND
    (
      -- 23160 = 06:26:00, 31140 = 08:39:00
      STOP_TIME >= 23160 AND STOP_TIME <= 31140 OR
      -- 57600 = 16:00:00, 66780 = 18:33:00
      STOP_TIME >= 57600 AND STOP_TIME <= 66780
    )
  GROUP BY
    SERVICE_DATE,
    ROUTE_NUMBER,
    LOCATION_ID,
    STOP_TIME
), diffs AS (
  SELECT
    TRUNC((t2.arrive_time - t1.arrive_time) / 60)::int AS
    prediction_diff,
    count(*) AS opservations
  FROM
    feb_data AS t1
    JOIN stop_event_avg_delay_dow_class AS t2
      ON  t1.ROUTE_NUMBER = t2.ROUTE_NUMBER AND
        t1.LOCATION_ID = t2.LOCATION_ID AND
```

```
        t1.STOP_TIME = t2.STOP_TIME AND
        t1.dow_class = t2.dow_class
    GROUP BY
      prediction_diff
)
SELECT
    CASE
      WHEN abs(prediction_diff) > 3 THEN 'others'
      ELSE prediction_diff::text
    END AS prediction_diffs,
    SUM(opservations) AS opservations
FROM
    diffs
GROUP BY
    prediction_diffs
ORDER BY
    prediction_diffs;
```

STMT 27: Finally, we compare Model 1 v2 and Model 2 v2 to the arrival times of a couple of routes at a given stop in real time (at the time this query was executed). STMT 23 to 26 give us general statistics on how accurate the model predictions are. Here we test those predictions in real time.

```
-- STMT: 27
-- Comparison between model 1 and model 2 predictions
SELECT
    t1.ROUTE_NUMBER,
    t1.LOCATION_ID,
    t1.STOP_TIME as stop_t_sec,
    t1.STOP_TIME * interval '1 sec' AS stop_time,
    t1.arrive_time * interval '1 sec' AS model1_pred_arrival_time,
    t1.leave_time * interval '1 sec' AS model1_pred_leave_time,
    t2.arrive_time * interval '1 sec' AS model2_pred_arrival_time,
```

```
  t2.leave_time * interval '1 sec' AS model2_pred_leave_time
FROM
  stop_event_avg_delay t1
  JOIN stop_event_avg_delay_dow_class t2
    ON
      t1.route_number = t2.route_number AND
      t1.location_id = t2.location_id AND
      t1.stop_time = t2.stop_time
WHERE
  t1.day_of_week = 5 AND
  t2.dow_class = 'D' AND
  t1.route_number in (76,78) AND
  t1.location_id = 2285
ORDER BY
  LOCATION_ID, STOP_TIME;
```

A.4 THE JSQL$_E$-EQUIVALENT ANALYSIS

The following is the equivalent analysis using jSQL$_e$ in the jSQL query language.

```
stop_events = IMPORT 'stop_events';

-- STMT: 1
route58_stop910 = SELECT stop_events WHERE
  service_date == '2018-12-10' AND
  route_number == 58 and location_id == 910;
route58_stop910_ordered = ORDER route58_stop910 BY arrive_time;


-- STMT: 2
stop9821 = SELECT stop_events WHERE
  service_date == '2018-12-10' AND
  location_id == 9821;
```

```
distinct_routes_at_stop9821 = DISTINCT stop9821 ON route_number;

-- STMT: 3
unique_stops = GROUP stop_events AS group ON
  service_date, route_number, location_id, stop_time;
unique_stops_count = AGGREGATE unique_stops ON group WITH
  count(*) AS occurances;
duplicates = SELECT unique_stops_count WHERE occurances > 1;


-- STMT: 4
route58_loc12790 = SELECT stop_events WHERE
  service_date == '2018-12-02' AND
  route_number == 58 AND
  location_id == 12790 AND
  stop_time == 38280;


-- STMT: 5
stop9818 = SELECT stop_events WHERE
  service_date == '2018-12-10' AND
  location_id == 9818;
distinct_routes_at_stop9818 = DISTINCT stop9818 ON route_number;


-- STMT: 6
stop_events_with_dow = PROJECT stop_events ADD
  extract('dow', service_date) AS day_of_week,
  ((arrive_time - stop_time) / 60)::int * 60 AS delay,
  CASE
    WHEN extract('dow', service_date) IN (1,2,3,4,5) THEN 'D'
    WHEN extract('dow', service_date) == 0 THEN 'U'
```

```
    ELSE 'S'
  END AS dow_class;
stop_events_with_dow_group = GROUP stop_events_with_dow AS group ON
  day_of_week, route_number, location_id, stop_time, delay;
stop_events_with_dow_histogram = AGGREGATE stop_events_with_dow_group
    ON group
  WITH count(*) AS num_of_observations;


-- STMT: 7
model1_v1_avg_delay_per_dow = SELECT stop_events_with_dow WHERE
  service_date BETWEEN ['2018-11-01', '2018-12-15') OR
  service_date BETWEEN ['2019-01-10', '2019-02-01');
model1_v1_avg_delay_per_dow_group = GROUP model1_v1_avg_delay_per_dow
    AS group ON
  day_of_week, route_number, location_id, stop_time;
model1_v1_agg = AGGREGATE model1_v1_avg_delay_per_dow_group ON group
    WITH
  AVG(arrive_time - stop_time) AS avg_delay_raw,
  count(*) AS num_of_observations;
model1_v1_proj = PROJECT model1_v1_agg ADD avg_delay_raw::int AS
    avg_delay;
model1_v1 = PROJECT model1_v1_proj EXCLUDE group;

-- STMT: 8
model1_v2_select_base_data = SELECT stop_events_with_dow WHERE
  (service_date BETWEEN ['2018-12-01', '2018-12-15') OR
  service_date BETWEEN ['2019-01-10', '2019-02-01')) AND
  route_number != 0;
model1_v2_select_base_data_with_delay = PROJECT
    model1_v2_select_base_data REPLACE
  CASE
```

```
    WHEN ABS(arrive_time - stop_time) <= ABS(leave_time - stop_time)
      THEN arrive_time - stop_time
    ELSE leave_time - stop_time
  END AS delay;
model1_v2_select_base_data_group = GROUP
    model1_v2_select_base_data_with_delay AS group ON
  service_date, day_of_week, route_number, location_id, stop_time;
model1_v2_cleaned_base_data = AGGREGATE REF
    model1_v2_select_base_data_group ON group WITH
  min(delay);
model1_v2_base_model_group = GROUP model1_v2_cleaned_base_data AS group
     ON
  day_of_week, route_number, location_id, stop_time;
model1_v2_base_model = AGGREGATE model1_v2_base_model_group ON group
  WITH STD(delay) AS std_delay,
  AVG(delay) AS avg_delay;
model1_v2_final_res_join = JOIN LEFT
  model1_v2_base_model WITH PREFIX t2_ , model1_v2_cleaned_base_data
    WITH PREFIX t1_ ON
  model1_v2_base_model.day_of_week == model1_v2_cleaned_base_data.
    day_of_week AND
  model1_v2_base_model.route_number == model1_v2_cleaned_base_data.
    route_number AND
  model1_v2_base_model.location_id == model1_v2_cleaned_base_data.
    location_id AND
  model1_v2_base_model.stop_time == model1_v2_cleaned_base_data.
    stop_time AND
  ABS(model1_v2_cleaned_base_data.delay) <= ABS(model1_v2_base_model.
    avg_delay) +  model1_v2_base_model.std_delay;
model1_v2_final_res_group = GROUP model1_v2_final_res_join AS group ON
  t2_day_of_week, t2_route_number, t2_location_id, t2_stop_time,
    t2_avg_delay;
```

```
model1_v2_final_res_agg = AGGREGATE model1_v2_final_res_group ON group
    WITH
  AVG(t1_delay) AS delay;
model1_v2 = PROJECT model1_v2_final_res_agg WITH
  t2_day_of_week AS day_of_week,
  t2_route_number AS route_number,
  t2_location_id AS location_id,
  t2_stop_time AS stop_time,
  IFNULL(delay, t2_avg_delay)::int AS avg_delay;


-- STMT: 9
model1_v2_compare_sel_route = SELECT model1_v2 WHERE
  route_number == 78;
model1_v2_compare_sel_dow_tue = SELECT model1_v2_compare_sel_route
    WHERE
  day_of_week == 2;
model1_v2_compare_sel_dow_wed = SELECT model1_v2_compare_sel_route
    WHERE
  day_of_week == 3;
model1_v2_compare_join = JOIN
  model1_v2_compare_sel_dow_tue WITH PREFIX t1_,
    model1_v2_compare_sel_dow_wed WITH PREFIX t2_ ON
  model1_v2_compare_sel_dow_tue.location_id ==
    model1_v2_compare_sel_dow_wed.location_id AND
  model1_v2_compare_sel_dow_tue.stop_time ==
    model1_v2_compare_sel_dow_wed.stop_time;
model1_v2_compare_project = PROJECT model1_v2_compare_join WITH
  t1_route_number AS route_number,
  t1_location_id AS location_id,
  t1_stop_time AS stop_time,
  (t1_avg_delay / 60)::int AS dow1_delay,
```

```
  (t2_avg_delay / 60)::int AS dow2_delay;
model1_v2_compare = ORDER model1_v2_compare_project BY location_id,
    stop_time;


-- STMT: 10
model2_v2_select_base_data_group = GROUP
    model1_v2_select_base_data_with_delay AS group ON
  service_date, dow_class, route_number, location_id, stop_time;
model2_v2_cleaned_base_data = AGGREGATE REF
    model2_v2_select_base_data_group ON group WITH
  min(delay);
model2_v2_base_model_group = GROUP model2_v2_cleaned_base_data AS group
     ON
  dow_class, route_number, location_id, stop_time;
model2_v2_base_model = AGGREGATE model2_v2_base_model_group ON group
  WITH STD(delay) AS std_delay,
  AVG(delay) AS avg_delay;
model2_v2_final_res_join = JOIN LEFT
  model2_v2_base_model WITH PREFIX t2_ , model2_v2_cleaned_base_data
    WITH PREFIX t1_ ON
  model2_v2_base_model.dow_class == model2_v2_cleaned_base_data.
    dow_class AND
  model2_v2_base_model.route_number == model2_v2_cleaned_base_data.
    route_number AND
  model2_v2_base_model.location_id == model2_v2_cleaned_base_data.
    location_id AND
  model2_v2_base_model.stop_time == model2_v2_cleaned_base_data.
    stop_time AND
  ABS(model2_v2_cleaned_base_data.delay) <= ABS(model2_v2_base_model.
    avg_delay) + model2_v2_base_model.std_delay;
model2_v2_final_res_group = GROUP model2_v2_final_res_join AS group ON
```

```
  t2_dow_class, t2_route_number, t2_location_id, t2_stop_time,
    t2_avg_delay;
model2_v2_final_res_agg = AGGREGATE model2_v2_final_res_group ON group
    WITH
  AVG(t1_delay) AS delay;
model2_v2 = PROJECT model2_v2_final_res_agg WITH
  t2_dow_class AS dow_class,
  t2_route_number AS route_number,
  t2_location_id AS location_id,
  t2_stop_time AS stop_time,
  IFNULL(delay, t2_avg_delay)::int AS avg_delay;


-- STMT: 11
model2_v2_2_avg_delay_per_dow_class = SELECT stop_events_with_dow WHERE
  service_date BETWEEN ['2018-11-01', '2019-02-01');
model2_v2_2_avg_delay_per_dow_group = GROUP
    model2_v2_2_avg_delay_per_dow_class AS group ON
  dow_class, route_number, location_id, stop_time;
model2_v2_2_agg = AGGREGATE model2_v2_2_avg_delay_per_dow_group ON
    group WITH
  AVG(arrive_time - stop_time) AS avg_delay_raw,
  count(*) AS num_of_observations;
model2_v2_2 = PROJECT model2_v2_2_agg ADD avg_delay_raw::int AS
    avg_delay;
model2_v2_2_proj = PROJECT model2_v2_2 EXCLUDE group;

-- STMT: 12
compare_v2_m1_m2_sel_m1 = SELECT model1_v2 WHERE
  day_of_week == 5 AND
  route_number in (76, 78) AND
  location_id == 2285;
```

```
compare_v2_m1_m2_sel_m2 = SELECT model2_v2 WHERE
  dow_class == 'D';
compare_v2_m1_m2_join = JOIN
  compare_v2_m1_m2_sel_m1 WITH PREFIX t1_, compare_v2_m1_m2_sel_m2 WITH
     PREFIX t2_ ON
  compare_v2_m1_m2_sel_m1.route_number == compare_v2_m1_m2_sel_m2.
     route_number AND
  compare_v2_m1_m2_sel_m1.location_id == compare_v2_m1_m2_sel_m2.
     location_id AND
  compare_v2_m1_m2_sel_m1.stop_time == compare_v2_m1_m2_sel_m2.
     stop_time;
compare_v2_m1_m2_project = PROJECT compare_v2_m1_m2_join WITH
  t1_route_number AS route_number,
  t1_location_id AS location_id,
  t1_stop_time AS stop_time,
  (t1_avg_delay / 60)::int AS dow1_delay,
  (t2_avg_delay / 60)::int AS dow2_delay;
compare_v2_m1_m2 = ORDER compare_v2_m1_m2_project BY location_id,
     stop_time;

-- STMT: 13
baseline_l1 = SELECT stop_events_with_dow WHERE
  service_date BETWEEN ['2019-02-01', '2019-03-01');
baseline_l2 = GROUP baseline_l1 AS group ON delay;
baseline_l3 = AGGREGATE baseline_l2 ON group WITH COUNT(*) AS
     observations;
baseline_l4 = PROJECT baseline_l3 ADD
  CASE WHEN ABS(delay) > 5 THEN 'others' ELSE delay::text END AS
     delay_diffs;
baseline_l5 = GROUP baseline_l4 AS group ON delay_diffs;
baseline_l6 = AGGREGATE baseline_l5 ON group WITH
  SUM(observations) AS observations;
```

```
baseline_l7 = PROJECT baseline_l6 WITH delay_diffs, observations;
baseline_l8 = ORDER baseline_l7 BY delay_diffs;

-- STMT: 14
baseline_rush_hour_l1 = SELECT baseline_l1 WHERE
  stop_time BETWEEN [23160, 31140] OR
  stop_time BETWEEN [57600, 66780];
baseline_rush_hour_l2 = GROUP baseline_rush_hour_l1 AS group ON delay;
baseline_rush_hour_l3 = AGGREGATE baseline_rush_hour_l2 ON group WITH
  COUNT(*) AS observations;
baseline_rush_hour_l4 = PROJECT baseline_rush_hour_l3 WITH delay,
    observations;
baseline_rush_hour_l5 = ORDER baseline_rush_hour_l4 BY observations
    DESC;

-- STMT: 15
predicting_feb_arrival_l1 = SELECT baseline_l1 WHERE
  route_number != 0 AND
  schedule_status != 6;
predicting_feb_arrival_l2 = PROJECT predicting_feb_arrival_l1 ADD
  (CASE
    WHEN ABS(arrive_time - stop_time) <= ABS(leave_time - stop_time)
      THEN arrive_time - stop_time
    ELSE leave_time - stop_time
  END / 60)::int AS actual_delay_in_min;
predicting_feb_arrival_l3 = JOIN
  predicting_feb_arrival_l2 WITH PREFIX t1_, model1_v2 WITH PREFIX t2_
    ON
  predicting_feb_arrival_l2.route_number == model1_v2.route_number AND
  predicting_feb_arrival_l2.location_id == model1_v2.location_id AND
  predicting_feb_arrival_l2.stop_time == model1_v2.stop_time AND
  predicting_feb_arrival_l2.day_of_week == model1_v2.day_of_week;
```

```
predicting_feb_arrival_l4 = PROJECT predicting_feb_arrival_l3 ADD
  (t2_avg_delay / 60)::int - t1_actual_delay_in_min AS delay_diff;
predicting_feb_arrival_l5 = GROUP predicting_feb_arrival_l4 AS group ON
    delay_diff;
predicting_feb_arrival_l6 = AGGREGATE predicting_feb_arrival_l5 ON
    group WITH
  COUNT(*) AS observations;
predicting_feb_arrival_l7 = PROJECT predicting_feb_arrival_l6 ADD
  CASE
    WHEN ABS(delay_diff) > 3 THEN 'others'
    ELSE delay_diff::text
  END AS delay_diffs;
predicting_feb_arrival_l8 = GROUP predicting_feb_arrival_l7 AS group ON
    delay_diffs;
predicting_feb_arrival_l9 = AGGREGATE predicting_feb_arrival_l8 ON
    group WITH
  SUM(observations) AS observations;
predicting_feb_arrival_l10 = PROJECT predicting_feb_arrival_l9 EXCLUDE
    group;
predicting_feb_arrival_l11 = ORDER predicting_feb_arrival_l10 BY
    delay_diffs;


-- STMT: 16
predicting_feb_arrival_rush_hr_l1 = SELECT baseline_l1 WHERE
  stop_time BETWEEN [23160, 31140] OR
  stop_time BETWEEN [57600, 66780];
predicting_feb_arrival_rush_hr_l2 = PROJECT
    predicting_feb_arrival_rush_hr_l1 ADD
  (delay / 60)::int AS actual_delay_in_min;
predicting_feb_arrival_rush_hr_l3 = JOIN
```

```
  predicting_feb_arrival_rush_hr_l2 WITH PREFIX t1_, model1_v2 WITH
    PREFIX t2_ ON
  predicting_feb_arrival_rush_hr_l2.route_number == model1_v2.
    route_number AND
  predicting_feb_arrival_rush_hr_l2.location_id == model1_v2.
    location_id AND
  predicting_feb_arrival_rush_hr_l2.stop_time == model1_v2.stop_time
    AND
  predicting_feb_arrival_rush_hr_l2.day_of_week == model1_v2.
    day_of_week;
predicting_feb_arrival_rush_hr_l4 = PROJECT
    predicting_feb_arrival_rush_hr_l3 ADD
  (t2_avg_delay / 60)::int - t1_actual_delay_in_min AS delay_diff;
predicting_feb_arrival_rush_hr_l5 = GROUP
    predicting_feb_arrival_rush_hr_l4 AS group ON delay_diff;
predicting_feb_arrival_rush_hr_l6 = AGGREGATE
    predicting_feb_arrival_rush_hr_l5 ON group WITH
  COUNT(*) AS observations;
predicting_feb_arrival_rush_hr_l7 = PROJECT
    predicting_feb_arrival_rush_hr_l6 EXCLUDE group;
predicting_feb_arrival_rush_hr_l8 = ORDER
    predicting_feb_arrival_rush_hr_l7 BY observations DESC;

-- STMT: 17
predicting_feb_arrival_dow_class_l1 = JOIN
  predicting_feb_arrival_l2 WITH PREFIX t1_, model2_v2_2_proj WITH
    PREFIX t2_ ON
  predicting_feb_arrival_l2.route_number == model2_v2_2_proj.
    route_number AND
  predicting_feb_arrival_l2.location_id == model2_v2_2_proj.location_id
     AND
  predicting_feb_arrival_l2.stop_time == model2_v2_2_proj.stop_time AND
```

```
    predicting_feb_arrival_l2.dow_class == model2_v2_2_proj.dow_class;
predicting_feb_arrival_dow_class_l2 = PROJECT
    predicting_feb_arrival_dow_class_l1 ADD
    (t2_avg_delay / 60)::int - t1_actual_delay_in_min AS delay_diff;
predicting_feb_arrival_dow_class_l3 = GROUP
    predicting_feb_arrival_dow_class_l2 AS group ON delay_diff;
predicting_feb_arrival_dow_class_l4 = AGGREGATE
    predicting_feb_arrival_dow_class_l3 ON group WITH
    COUNT(*) AS observations;
predicting_feb_arrival_dow_class_l5 = PROJECT
    predicting_feb_arrival_dow_class_l4 ADD
    CASE
        WHEN ABS(delay_diff) > 3 THEN 'others'
        ELSE delay_diff::text
    END AS delay_diffs;
predicting_feb_arrival_dow_class_l6 = GROUP
    predicting_feb_arrival_dow_class_l5 AS group ON delay_diffs;
predicting_feb_arrival_dow_class_l7 = AGGREGATE
    predicting_feb_arrival_dow_class_l6 ON group WITH
    SUM(observations) AS observations;
predicting_feb_arrival_dow_class_l8 = PROJECT
    predicting_feb_arrival_dow_class_l7 EXCLUDE group;
predicting_feb_arrival_dow_class_l9 = ORDER
    predicting_feb_arrival_dow_class_l8 BY delay_diffs;

-- STMT: 18
predicting_feb_arrival_rush_hr_dow_class_l1 =
    JOIN predicting_feb_arrival_rush_hr_l2 WITH PREFIX t1_,
    model2_v2_2_proj WITH PREFIX t2_ ON
    predicting_feb_arrival_rush_hr_l2.route_number == model2_v2_2_proj.
    route_number AND
```

```
  predicting_feb_arrival_rush_hr_l2.location_id == model2_v2_2_proj.
    location_id AND
  predicting_feb_arrival_rush_hr_l2.stop_time == model2_v2_2_proj.
    stop_time AND
  predicting_feb_arrival_rush_hr_l2.dow_class == model2_v2_2_proj.
    dow_class;
predicting_feb_arrival_rush_hr_dow_class_l2 = PROJECT
    predicting_feb_arrival_rush_hr_dow_class_l1 ADD
  (t2_avg_delay / 60)::int - t1_actual_delay_in_min AS delay_diff;
predicting_feb_arrival_rush_hr_dow_class_l3 = GROUP
    predicting_feb_arrival_rush_hr_dow_class_l2 AS group ON delay_diff;
predicting_feb_arrival_rush_hr_dow_class_l4 = AGGREGATE
    predicting_feb_arrival_rush_hr_dow_class_l3 ON group WITH
  COUNT(*) AS observations;
predicting_feb_arrival_rush_hr_dow_class_l5 = PROJECT
    predicting_feb_arrival_rush_hr_dow_class_l4 EXCLUDE group;
predicting_feb_arrival_rush_hr_dow_class_l6 = ORDER
    predicting_feb_arrival_rush_hr_dow_class_l5 BY observations DESC;


-- STMT: 19
model1_v3_l1 = GROUP model1_v2_select_base_data AS group ON
  service_date, route_number, location_id, stop_time;
model1_v3_l2 = AGGREGATE model1_v3_l1 ON group WITH
  MAX(arrive_time) AS arrive_time,
  MAX(leave_time) AS leave_time;
model1_v3_l3 = PROJECT model1_v3_l2 ADD
  extract('dow', service_date) AS day_of_week;
model1_v3_l4 = GROUP model1_v3_l3 AS group ON
  day_of_week, route_number, location_id, stop_time;
model1_v3_l5 = AGGREGATE model1_v3_l4 ON group WITH
  STD(arrive_time) AS std_arrive_time,
```

```
  AVG(arrive_time) AS avg_arrive_time,
  STD(leave_time) AS std_leave_time,
  AVG(leave_time) AS avg_leave_time;
model1_v3_l6 = JOIN
  model1_v3_l3 WITH PREFIX t1_, model1_v3_l5 WITH PREFIX t2_ ON
  model1_v3_l5.day_of_week == model1_v3_l3.day_of_week AND
  model1_v3_l5.route_number == model1_v3_l3.route_number AND
  model1_v3_l5.location_id == model1_v3_l3.location_id AND
  model1_v3_l5.stop_time == model1_v3_l3.stop_time;
model1_v3_l7 = GROUP model1_v3_l6 AS group ON
  t2_day_of_week, t2_route_number, t2_location_id,
  t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time;
model1_v3_l8 = AGGREGATE model1_v3_l7 ON group WITH
  AVG(t1_arrive_time) WHERE(
    abs(t1_arrive_time) <= abs(t2_avg_arrive_time) + t2_std_arrive_time
  ) AS avg_arrive_time,
  AVG(t1_leave_time) WHERE(
    abs(t1_leave_time) <= abs(t2_avg_leave_time) + t2_std_leave_time
  ) AS avg_leave_time;
model1_v3_l9 = PROJECT model1_v3_l8 WITH
  t2_day_of_week AS day_of_week,
  t2_route_number AS route_number,
  t2_location_id AS location_id,
  t2_stop_time AS stop_time,
  IFNULL(avg_arrive_time, t2_avg_arrive_time)::int AS arrive_time,
  IFNULL(avg_leave_time,t2_avg_leave_time)::int AS leave_time;


-- STMT: 20
model2_v3_l1 = PROJECT model1_v3_l3 ADD
  CASE
    WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
```

```
    WHEN day_of_week == 0 THEN 'U'
    ELSE 'S'
  END AS dow_class;
model2_v3_l2 = GROUP model2_v3_l1 AS group ON
  dow_class, route_number, location_id, stop_time;
model2_v3_l3 = AGGREGATE model2_v3_l2 ON group WITH
  STD(arrive_time) AS std_arrive_time,
  AVG(arrive_time) AS avg_arrive_time,
  STD(leave_time) AS std_leave_time,
  AVG(leave_time) AS avg_leave_time;
model2_v3_l4 = JOIN
  model2_v3_l1 WITH PREFIX t1_, model2_v3_l3 WITH PREFIX t2_ ON
  model2_v3_l1.dow_class == model2_v3_l3.dow_class AND
  model2_v3_l1.route_number == model2_v3_l3.route_number AND
  model2_v3_l1.location_id == model2_v3_l3.location_id AND
  model2_v3_l1.stop_time == model2_v3_l3.stop_time;
model2_v3_l5 = GROUP model2_v3_l4 AS group ON
  t2_dow_class, t2_route_number, t2_location_id,
  t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time;
model2_v3_l6 = AGGREGATE model2_v3_l5 ON group WITH
  AVG(t1_arrive_time) WHERE(
    abs(t1_arrive_time) <= abs(t2_avg_arrive_time) + t2_std_arrive_time
  ) AS avg_arrive_time,
  AVG(t1_leave_time) WHERE(
    abs(t1_leave_time) <= abs(t2_avg_leave_time) + t2_std_leave_time
  ) AS avg_leave_time;
model2_v3_l7 = PROJECT model2_v3_l6 WITH
  t2_dow_class AS dow_class,
  t2_route_number AS route_number,
  t2_location_id AS location_id,
  t2_stop_time AS stop_time,
  IFNULL(avg_arrive_time, t2_avg_arrive_time)::int AS arrive_time,
```

```
  IFNULL(avg_leave_time,t2_avg_leave_time)::int AS leave_time;

-- STMT: 21
baseline_v2_l1 = SELECT baseline_l1 WHERE route_number != 0;
baseline_v2_l2 = GROUP baseline_v2_l1 AS group ON
  service_date, route_number, location_id, stop_time;
baseline_v2_l3 = AGGREGATE baseline_v2_l2 ON group WITH
  MIN(arrive_time) AS arrive_time,
  MAX(leave_time) AS leave_time;
baseline_v2_l4 = PROJECT baseline_v2_l3 WITH
  (CASE
    WHEN ABS(arrive_time - stop_time) <= ABS(leave_time - 30 -
    stop_time)
      THEN arrive_time - stop_time
    ELSE leave_time - 30 - stop_time
  END / 60)::int AS prediction_diff;
baseline_v2_l5 = PROJECT baseline_v2_l4 WITH
  CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE prediction_diff::text
  END AS prediction_diffs;
baseline_v2_l6 = GROUP baseline_v2_l5 AS group ON prediction_diffs;
baseline_v2_l7 = AGGREGATE baseline_v2_l6 ON group WITH
  COUNT(*) AS observations;
baseline_v2_l8 = PROJECT baseline_v2_l7 EXCLUDE group;
baseline_v2_l9 = ORDER baseline_v2_l8 BY prediction_diffs;

-- STMT: 22
baseline_v2_rush_hour_l1 = SELECT baseline_rush_hour_l1 WHERE
  route_number != 0;
baseline_v2_rush_hour_l2 = GROUP baseline_v2_rush_hour_l1 AS group ON
  service_date, route_number, location_id, stop_time;
```

```
baseline_v2_rush_hour_l3 = AGGREGATE baseline_v2_rush_hour_l2 ON group
    WITH
  MIN(arrive_time) AS arrive_time,
  MAX(leave_time) AS leave_time;
baseline_v2_rush_hour_l4 = PROJECT baseline_v2_rush_hour_l3 WITH
  (CASE
    WHEN ABS(arrive_time - stop_time) <= ABS(leave_time - 30 -
    stop_time)
      THEN arrive_time - stop_time
    ELSE leave_time - 30 - stop_time
  END / 60)::int AS prediction_diff;
baseline_v2_rush_hour_l5 = PROJECT baseline_v2_rush_hour_l4 WITH
  CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE prediction_diff::text
  END AS prediction_diffs;
baseline_v2_rush_hour_l6 = GROUP baseline_v2_rush_hour_l5 AS group ON
    prediction_diffs;
baseline_v2_rush_hour_l7 = AGGREGATE baseline_v2_rush_hour_l6 ON group
    WITH
  COUNT(*) AS observations;
baseline_v2_rush_hour_l8 = PROJECT baseline_v2_rush_hour_l7 EXCLUDE
    group;
baseline_v2_rush_hour_l9 = ORDER baseline_v2_rush_hour_l8 BY
    prediction_diffs;

-- STMT: 23
comp_predic_v2_l1 = PROJECT baseline_v2_l3 ADD
  extract('dow', service_date) AS day_of_week;
comp_predic_v2_l2 = JOIN
  comp_predic_v2_l1 WITH PREFIX t1_, model1_v3_l9 WITH PREFIX t2_ ON
  comp_predic_v2_l1.route_number == model1_v3_l9.route_number AND
```

```
  comp_predic_v2_l1.location_id == model1_v3_l9.location_id AND
  comp_predic_v2_l1.stop_time == model1_v3_l9.stop_time AND
  comp_predic_v2_l1.day_of_week == model1_v3_l9.day_of_week;
comp_predic_v2_l3 = PROJECT comp_predic_v2_l2 ADD
  ((t2_arrive_time - t1_arrive_time) / 60)::int AS prediction_diff;
comp_predic_v2_l4 = GROUP comp_predic_v2_l3 AS group ON prediction_diff
    ;
comp_predic_v2_l5 = AGGREGATE comp_predic_v2_l4 ON group WITH
  COUNT(*) AS observations;
comp_predic_v2_l6 = PROJECT comp_predic_v2_l5 ADD
  CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE prediction_diff::text
  END AS prediction_diffs;
comp_predic_v2_l7 = GROUP comp_predic_v2_l6 AS group ON
    prediction_diffs;
comp_predic_v2_l8 = AGGREGATE comp_predic_v2_l7 ON group WITH
  SUM(observations) AS observations;
comp_predic_v2_l9 = PROJECT comp_predic_v2_l8 EXCLUDE group;
comp_predic_v2_l10 = ORDER comp_predic_v2_l9 BY prediction_diffs;

-- STMT: 24
comp_predic_v2_rush_hour_l1 = PROJECT baseline_v2_rush_hour_l3 ADD
  extract('dow', service_date) AS day_of_week;
comp_predic_v2_rush_hour_l2 = JOIN
  comp_predic_v2_rush_hour_l1 WITH PREFIX t1_, model1_v3_l9 WITH PREFIX
    t2_ ON
  comp_predic_v2_rush_hour_l1.route_number == model1_v3_l9.route_number
    AND
  comp_predic_v2_rush_hour_l1.location_id == model1_v3_l9.location_id
    AND
  comp_predic_v2_rush_hour_l1.stop_time == model1_v3_l9.stop_time AND
```

```
  comp_predic_v2_rush_hour_l1.day_of_week == model1_v3_l9.day_of_week;
comp_predic_v2_rush_hour_l3 = PROJECT comp_predic_v2_rush_hour_l2 ADD
  ((t2_arrive_time - t1_arrive_time) / 60)::int AS prediction_diff;
comp_predic_v2_rush_hour_l4 = GROUP comp_predic_v2_rush_hour_l3 AS
    group ON
  prediction_diff;
comp_predic_v2_rush_hour_l5 = AGGREGATE comp_predic_v2_rush_hour_l4 ON
    group WITH
  COUNT(*) AS observations;
comp_predic_v2_rush_hour_l6 = PROJECT comp_predic_v2_rush_hour_l5 ADD
  CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE prediction_diff::text
  END AS prediction_diffs;
comp_predic_v2_rush_hour_l7 = GROUP comp_predic_v2_rush_hour_l6 AS
    group ON
  prediction_diffs;
comp_predic_v2_rush_hour_l8 = AGGREGATE comp_predic_v2_rush_hour_l7 ON
    group WITH
  SUM(observations) AS observations;
comp_predic_v2_rush_hour_l9 = PROJECT comp_predic_v2_rush_hour_l8
    EXCLUDE group;
comp_predic_v2_rush_hour_l10 = ORDER comp_predic_v2_rush_hour_l9 BY
    prediction_diffs;

-- STMT: 25
comp_predic_v3_l1 = PROJECT comp_predic_v2_l1 ADD
  CASE
    WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
    WHEN day_of_week == 0 THEN 'U'
    ELSE 'S'
  END AS dow_class;
```

```
comp_predic_v3_l2 = JOIN
  comp_predic_v3_l1 WITH PREFIX t1_, model2_v3_l7 WITH PREFIX t2_ ON
  comp_predic_v3_l1.route_number == model2_v3_l7.route_number AND
  comp_predic_v3_l1.location_id == model2_v3_l7.location_id AND
  comp_predic_v3_l1.stop_time == model2_v3_l7.stop_time AND
  comp_predic_v3_l1.dow_class == model2_v3_l7.dow_class;
comp_predic_v3_l3 = PROJECT comp_predic_v3_l2 ADD
  ((t2_arrive_time - t1_arrive_time) / 60)::int AS prediction_diff;
comp_predic_v3_l4 = GROUP comp_predic_v3_l3 AS group ON prediction_diff
    ;
comp_predic_v3_l5 = AGGREGATE comp_predic_v3_l4 ON group WITH
  COUNT(*) AS observations;
comp_predic_v3_l6 = PROJECT comp_predic_v3_l5 ADD
  CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE prediction_diff::text
  END AS prediction_diffs;
comp_predic_v3_l7 = GROUP comp_predic_v3_l6 AS group ON
    prediction_diffs;
comp_predic_v3_l8 = AGGREGATE comp_predic_v3_l7 ON group WITH
  SUM(observations) AS observations;
comp_predic_v3_l9 = PROJECT comp_predic_v3_l8 EXCLUDE group;
comp_predic_v3_l10 = ORDER comp_predic_v3_l9 BY prediction_diffs;

-- STMT: 26
comp_predic_v3_rush_hour_l1 = PROJECT comp_predic_v2_rush_hour_l1 ADD
  CASE
    WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
    WHEN day_of_week == 0 THEN 'U'
    ELSE 'S'
  END AS dow_class;
comp_predic_v3_rush_hour_l2 = JOIN
```

```
  comp_predic_v3_rush_hour_l1 WITH PREFIX t1_, model2_v3_l7 WITH PREFIX
      t2_ ON
  comp_predic_v3_rush_hour_l1.route_number == model2_v3_l7.route_number
      AND
  comp_predic_v3_rush_hour_l1.location_id == model2_v3_l7.location_id
     AND
  comp_predic_v3_rush_hour_l1.stop_time == model2_v3_l7.stop_time AND
  comp_predic_v3_rush_hour_l1.dow_class == model2_v3_l7.dow_class;
comp_predic_v3_rush_hour_l3 = PROJECT comp_predic_v3_rush_hour_l2 ADD
  ((t2_arrive_time - t1_arrive_time) / 60)::int AS prediction_diff;
comp_predic_v3_rush_hour_l4 = GROUP comp_predic_v3_rush_hour_l3 AS
    group ON
  prediction_diff;
comp_predic_v3_rush_hour_l5 = AGGREGATE comp_predic_v3_rush_hour_l4 ON
    group WITH
  COUNT(*) AS observations;
comp_predic_v3_rush_hour_l6 = PROJECT comp_predic_v3_rush_hour_l5 ADD
  CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE prediction_diff::text
  END AS prediction_diffs;
comp_predic_v3_rush_hour_l7 = GROUP comp_predic_v3_rush_hour_l6 AS
    group ON
  prediction_diffs;
comp_predic_v3_rush_hour_l8 = AGGREGATE comp_predic_v3_rush_hour_l7 ON
    group WITH
  SUM(observations) AS observations;
comp_predic_v3_rush_hour_l9 = PROJECT comp_predic_v3_rush_hour_l8
    EXCLUDE group;
comp_predic_v3_rush_hour_l10 = ORDER comp_predic_v3_rush_hour_l9 BY
    prediction_diffs;

```

```
-- STMT: 27
comp_pred_model1_and_model2_l1 = JOIN
  model1_v3_l9 WITH PREFIX t1_, model2_v3_l7 WITH PREFIX t2_ ON
  model1_v3_l9.route_number == model2_v3_l7.route_number AND
  model1_v3_l9.location_id == model2_v3_l7.location_id AND
  model1_v3_l9.stop_time == model2_v3_l7.stop_time;
comp_pred_model1_and_model2_l2 = SELECT comp_pred_model1_and_model2_l1
    WHERE
  t1_day_of_week == 5 AND
  t2_dow_class == 'D' AND
  t1_route_number in (76, 78) AND
  t1_location_id == 2285;
comp_pred_model1_and_model2_l3 = PROJECT comp_pred_model1_and_model2_l2
    WITH
  t1_route_number AS route_number,
  t1_location_id AS location_id,
  t1_stop_time AS stop_time,
  t1_arrive_time AS model1_pred_arrival_time,
  t1_leave_time AS model1_pred_leave_time,
  t2_arrive_time AS model2_pred_arrival_time,
  t2_leave_time AS model2_pred_leave_time;
comp_pred_model1_and_model2_l4 = ORDER comp_pred_model1_and_model2_l3
    BY
  location_id, stop_time;
```

A.4.1 Min-Max Queries

The following are the min-max queries that we used to test data-access time[2] at the top of each of the 27 stacks in addition to stack 0 (the original data set).

[2]These queries are more about build time than just access time. Access time is the time it takes to access only the data, whereas build time is the time it takes to access the data in addition to processing it. Since the queries are about applying an `aggregate` operator, what we are measuring is the time it takes to access the data and perform the aggregations as well.

```
-- MIN/MAX QUERIES FOR EVERY STACK

-- STACK 0:
min_max_query0 = AGGREGATE stop_events WITH
  MIN(service_date) AS min_date,
  MAX(service_date) AS max_date;

-- STACK 1:
min_max_query1 = AGGREGATE route58_stop910_ordered WITH
  MIN(service_date) AS min_date,
  MAX(service_date) AS max_date;

-- STACK 2:
min_max_query2 = AGGREGATE distinct_routes_at_stop9821 WITH
  MIN(route_number) AS min_route_num,
  MAX(route_number) AS max_route_num;

-- STACK 3:
min_max_query3 = AGGREGATE duplicates WITH
  MIN(service_date) AS min_date,
  MAX(service_date) AS max_date;

-- STACK 4:
min_max_query4 = AGGREGATE route58_loc12790 WITH
  MIN(service_date) AS min_date,
  MAX(service_date) AS max_date;

-- STACK 5:
min_max_query5 = AGGREGATE distinct_routes_at_stop9818 WITH
  MIN(route_number) AS min_route_num,
  MAX(route_number) AS max_route_num;

```

```
-- STACK 6:
min_max_query6 = AGGREGATE stop_events_with_dow_histogram WITH
  MIN(stop_time) AS min_stop_time,
  MAX(stop_time) AS max_stop_time;

-- STACK 7:
min_max_query7 = AGGREGATE model1_v1 WITH
  MIN(stop_time) AS min_stop_time,
  MAX(stop_time) AS max_stop_time;

-- STACK 8:
min_max_query8 = AGGREGATE model1_v2 WITH
  MIN(stop_time) AS min_stop_time,
  MAX(stop_time) AS max_stop_time;

-- STACK 9:
min_max_query9 = AGGREGATE model1_v2_compare WITH
  MIN(stop_time) AS min_stop_time,
  MAX(stop_time) AS max_stop_time;

-- STACK 10:
min_max_query10 = AGGREGATE model2_v2 WITH
  MIN(stop_time) AS min_stop_time,
  MAX(stop_time) AS max_stop_time;

-- STACK 11:
min_max_query11 = AGGREGATE model2_v2_2_proj WITH
  MIN(stop_time) AS min_stop_time,
  MAX(stop_time) AS max_stop_time;

-- STACK 12:
min_max_query12 = AGGREGATE compare_v2_m1_m2 WITH
```

```
  MIN(stop_time) AS min_stop_time,
  MAX(stop_time) AS max_stop_time;

-- STACK 13:
min_max_query13 = AGGREGATE baseline_l8 WITH
  MIN(delay_diffs) AS min_delay_diffs,
  MAX(delay_diffs) AS max_delay_diffs;

-- STACK 14:
min_max_query14 = AGGREGATE baseline_rush_hour_l5 WITH
  MIN(delay) AS min_delay,
  MAX(delay) AS max_delay;

-- STACK 15:
min_max_query15 = AGGREGATE predicting_feb_arrival_l11 WITH
  MIN(delay_diffs) AS min_delay_diffs,
  MAX(delay_diffs) AS max_delay_diffs;

-- STACK 16:
min_max_query16 = AGGREGATE predicting_feb_arrival_rush_hr_l8 WITH
  MIN(delay_diff) AS min_delay_diff,
  MAX(delay_diff) AS max_delay_diff;

-- STACK 17:
min_max_query17 = AGGREGATE predicting_feb_arrival_dow_class_l9 WITH
  MIN(delay_diffs) AS min_delay_diffs,
  MAX(delay_diffs) AS max_delay_diffs;

-- STACK 18:
min_max_query18 = AGGREGATE predicting_feb_arrival_rush_hr_dow_class_l6
    WITH
  MIN(delay_diff) AS min_delay_diff,
```

```
  MAX(delay_diff) AS max_delay_diff;

-- STACK 19:
min_max_query19 = AGGREGATE model1_v3_l9 WITH
  MIN(stop_time) AS min_stop_time,
  MAX(stop_time) AS max_stop_time;

-- STACK 20:
min_max_query20 = AGGREGATE model2_v3_l7 WITH
  MIN(stop_time) AS min_stop_time,
  MAX(stop_time) AS max_stop_time;

-- STACK 21:
min_max_query21 = AGGREGATE baseline_v2_l9 WITH
  MIN(prediction_diffs) AS min_prediction_diffs,
  MAX(prediction_diffs) AS max_prediction_diffs;

-- STACK 22:
min_max_query22 = AGGREGATE baseline_v2_rush_hour_l9 WITH
  MIN(prediction_diffs) AS min_prediction_diffs,
  MAX(prediction_diffs) AS max_prediction_diffs;

-- STACK 23:
min_max_query23 = AGGREGATE comp_predic_v2_l10 WITH
  MIN(prediction_diffs) AS min_prediction_diffs,
  MAX(prediction_diffs) AS max_prediction_diffs;

-- STACK 24:
min_max_query24 = AGGREGATE comp_predic_v2_rush_hour_l10 WITH
  MIN(prediction_diffs) AS min_prediction_diffs,
  MAX(prediction_diffs) AS max_prediction_diffs;

```

```
-- STACK 25:
min_max_query25 = AGGREGATE comp_predic_v3_l10 WITH
  MIN(prediction_diffs) AS min_prediction_diffs,
  MAX(prediction_diffs) AS max_prediction_diffs;

-- STACK 26:
min_max_query26 = AGGREGATE comp_predic_v3_rush_hour_l10 WITH
  MIN(prediction_diffs) AS min_prediction_diffs,
  MAX(prediction_diffs) AS max_prediction_diffs;

-- STACK 27:
min_max_query27 = AGGREGATE comp_pred_model1_and_model2_l4 WITH
  MIN(stop_time) AS min_stop_time,
  MAX(stop_time) AS max_stop_time;
```

A.5 MYSQL-EQUIVALENT ANALYSIS

The following is the equivalent analysis using MySQL [18] with in-memory tables. The goal here is to materialize every possible intermediate result to simulate what we did with jSQL$_e$.

```
CREATE TABLE stop_events
(
    service_date date,
    leave_time integer,
    route_number integer,
    stop_time integer,
    arrive_time integer,
    location_id integer,
    schedule_status integer
) ENGINE = MEMORY;


```

```
-- STMT: 1
CREATE TABLE route58_stop910 ENGINE = MEMORY
SELECT * FROM stop_events
WHERE
    service_date = '2018-12-10' AND
    route_number = 58 AND
    location_id = 910;

CREATE TABLE route58_stop910_ordered ENGINE = MEMORY
SELECT * FROM route58_stop910 ORDER BY arrive_time;


-- STMT: 2
CREATE TABLE stop9821 ENGINE = MEMORY
SELECT * FROM stop_events
WHERE
    service_date = '2018-12-10' AND
    location_id = 9821;

CREATE TABLE distinct_routes_at_stop9821 ENGINE = MEMORY
SELECT DISTINCT route_number FROM stop9821;


-- STMT: 3
CREATE TABLE unique_stops_count ENGINE = MEMORY
SELECT
    service_date,
    route_number,
    location_id,
    stop_time,
    count(*) as occurances
FROM stop_events
```

```
45 GROUP BY
46     service_date, route_number,
47     location_id, stop_time;
48
49 CREATE TABLE duplicates ENGINE = MEMORY
50 SELECT * FROM unique_stops_count WHERE occurances > 1;
51
52
53 -- STMT: 4
54 CREATE TABLE route58_loc12790 ENGINE = MEMORY
55 SELECT * FROM stop_events
56 WHERE
57     service_date = '2018-12-02' AND
58     route_number = 58 AND
59     location_id = 12790 AND
60     stop_time = 38280;
61
62
63 -- STMT: 5
64 CREATE TABLE stop9818 ENGINE = MEMORY
65 SELECT * FROM stop_events
66 WHERE
67     service_date = '2018-12-10' AND
68     location_id = 9818;
69
70 CREATE TABLE distinct_routes_at_stop9818 ENGINE = MEMORY
71 SELECT DISTINCT route_number FROM stop9818;
72
73
74 -- STMT: 6
75 CREATE TABLE stop_events_with_dow ENGINE = MEMORY
76 SELECT
```

```
77      t1.*, DAYOFWEEK(service_date) - 1 AS day_of_week,
78      CAST(TRUNCATE((arrive_time - stop_time) / 60, 0) AS SIGNED INTEGER)
            * 60 AS delay,
79      CASE
80          WHEN DAYOFWEEK(service_date) - 1 IN (1, 2,3,4,5) THEN 'D'
81          WHEN DAYOFWEEK(service_date) - 1 = 1 THEN 'U'
82          ELSE 'S'
83      END AS dow_class
84  FROM stop_events AS t1;
85
86  CREATE TABLE stop_events_with_dow_histogram ENGINE = MEMORY
87  SELECT
88      day_of_week, route_number, location_id,
89      stop_time, delay,
90      count(*) AS num_of_observations
91  FROM stop_events_with_dow
92  GROUP BY
93      day_of_week, route_number,
94      location_id, stop_time, delay;
95
96
97  -- STMT: 7
98  CREATE TABLE model1_v1_avg_delay_per_dow ENGINE = MEMORY
99  SELECT *
100 FROM stop_events_with_dow
101 WHERE
102     service_date >= '2018-11-01' AND
103     service_date < '2018-12-15' OR
104     service_date >= '2019-01-10' AND
105     service_date < '2019-02-01';
106
107 CREATE TABLE model1_v1_agg ENGINE = MEMORY
```

```
SELECT
    day_of_week, route_number,
    location_id, stop_time,
    AVG(arrive_time - stop_time) AS avg_delay_raw,
    count(*) AS num_of_observations
FROM model1_v1_avg_delay_per_dow
GROUP BY
    day_of_week, route_number,
    location_id, stop_time;

CREATE TABLE model1_v1 ENGINE = MEMORY
SELECT
    t1.*,
    CAST(TRUNCATE(avg_delay_raw, 0) AS SIGNED INTEGER) AS avg_delay
FROM model1_v1_agg AS t1;


-- STMT: 8
CREATE TABLE model1_v2_select_base_data ENGINE = MEMORY
SELECT *
FROM stop_events_with_dow
WHERE
    (
        service_date >= '2018-12-01' AND
        service_date < '2018-12-15' OR
        service_date >= '2019-01-10' AND
        service_date < '2019-02-01'
    ) AND
    route_number <> 0;

CREATE TABLE model1_v2_select_base_data_with_delay ENGINE = MEMORY
SELECT
```

```
  service_date, leave_time, route_number,
    stop_time, arrive_time, location_id,
    schedule_status, day_of_week, dow_class,
    CASE
        WHEN ABS(arrive_time - stop_time) <= ABS(leave_time -
            stop_time)
            THEN arrive_time - stop_time
        ELSE leave_time - stop_time
    END AS delay
FROM model1_v2_select_base_data AS t1;

CREATE TABLE model1_v2_cleaned_base_data ENGINE = MEMORY
SELECT t1.*
FROM
    model1_v2_select_base_data_with_delay AS t1,
    (
        SELECT
            service_date, day_of_week,
            route_number, location_id, stop_time,
            min(delay) AS min_delay
        FROM model1_v2_select_base_data_with_delay
        GROUP BY
            service_date, day_of_week,
            route_number, location_id, stop_time
    ) AS t2
WHERE
    t1.delay = t2.min_delay AND
    t1.service_date = t2.service_date AND
    t1.day_of_week = t2.day_of_week AND
    t1.route_number = t2.route_number AND
    t1.location_id = t2.location_id AND
    t1.stop_time = t2.stop_time;
```

```
171
172 CREATE TABLE model1_v2_base_model ENGINE = MEMORY
173 SELECT
174     day_of_week, route_number,
175     location_id, stop_time,
176     STDDEV(delay) AS std_delay,
177     AVG(delay) AS avg_delay
178 FROM model1_v2_cleaned_base_data
179 GROUP BY
180     day_of_week, route_number,
181     location_id, stop_time;
182
183
184 CREATE TABLE model1_v2_final_res_join ENGINE = MEMORY
185 SELECT
186     model1_v2_base_model.day_of_week AS t2_day_of_week,
187     model1_v2_base_model.route_number AS t2_route_number,
188     model1_v2_base_model.location_id AS t2_location_id,
189     model1_v2_base_model.stop_time AS t2_stop_time,
190     model1_v2_base_model.std_delay AS t2_std_delay,
191     model1_v2_base_model.avg_delay AS t2_avg_delay,
192     model1_v2_cleaned_base_data.service_date AS t1_service_date,
193     model1_v2_cleaned_base_data.leave_time AS t1_leave_time,
194     model1_v2_cleaned_base_data.route_number AS t1_route_number,
195     model1_v2_cleaned_base_data.stop_time AS t1_stop_time,
196     model1_v2_cleaned_base_data.arrive_time AS t1_arrive_time,
197     model1_v2_cleaned_base_data.location_id AS t1_location_id,
198     model1_v2_cleaned_base_data.schedule_status AS t1_schedule_status,
199     model1_v2_cleaned_base_data.day_of_week AS t1_day_of_week,
200     model1_v2_cleaned_base_data.dow_class AS t1_dow_class,
201     model1_v2_cleaned_base_data.delay AS t1_delay
202 FROM
```

```
    model1_v2_base_model
    LEFT JOIN model1_v2_cleaned_base_data ON
        model1_v2_base_model.day_of_week = model1_v2_cleaned_base_data.
            day_of_week AND
        model1_v2_base_model.route_number = model1_v2_cleaned_base_data
            .route_number AND
        model1_v2_base_model.location_id = model1_v2_cleaned_base_data.
            location_id AND
        model1_v2_base_model.stop_time = model1_v2_cleaned_base_data.
            stop_time AND
        ABS(model1_v2_cleaned_base_data.delay) <= ABS(
            model1_v2_base_model.avg_delay) + model1_v2_base_model.
            std_delay;

CREATE TABLE model1_v2_final_res_agg ENGINE = MEMORY
SELECT
    t2_day_of_week, t2_route_number, t2_location_id,
    t2_stop_time, t2_avg_delay, AVG(t1_delay) AS delay
FROM model1_v2_final_res_join
GROUP BY
    t2_day_of_week, t2_route_number,
    t2_location_id, t2_stop_time, t2_avg_delay;

CREATE TABLE model1_v2 ENGINE = MEMORY
SELECT
    t2_day_of_week AS day_of_week,
    t2_route_number AS route_number,
    t2_location_id AS location_id,
    t2_stop_time AS stop_time,
    CAST(
        TRUNCATE(COALESCE(delay, t2_avg_delay), 0)
        AS SIGNED INTEGER
```

```
    ) AS avg_delay
FROM model1_v2_final_res_agg;


-- STMT: 9
CREATE TABLE model1_v2_compare_sel_route ENGINE = MEMORY
SELECT *
FROM model1_v2
WHERE route_number = 78;

CREATE TABLE model1_v2_compare_sel_dow_tue ENGINE = MEMORY
SELECT *
FROM model1_v2_compare_sel_route
WHERE day_of_week = 2;

CREATE TABLE model1_v2_compare_sel_dow_wed ENGINE = MEMORY
SELECT *
FROM model1_v2_compare_sel_route
WHERE day_of_week = 3;

CREATE TABLE model1_v2_compare_join ENGINE = MEMORY
SELECT
    model1_v2_compare_sel_dow_tue.day_of_week AS t1_day_of_week,
    model1_v2_compare_sel_dow_tue.route_number AS t1_route_number,
    model1_v2_compare_sel_dow_tue.location_id AS t1_location_id,
    model1_v2_compare_sel_dow_tue.stop_time AS t1_stop_time,
    model1_v2_compare_sel_dow_tue.avg_delay AS t1_avg_delay,
    model1_v2_compare_sel_dow_wed.day_of_week AS t2_day_of_week,
    model1_v2_compare_sel_dow_wed.route_number AS t2_route_number,
    model1_v2_compare_sel_dow_wed.location_id AS t2_location_id,
    model1_v2_compare_sel_dow_wed.stop_time AS t2_stop_time,
    model1_v2_compare_sel_dow_wed.avg_delay AS t2_avg_delay
```

```
FROM
    model1_v2_compare_sel_dow_tue
    JOIN model1_v2_compare_sel_dow_wed ON
        model1_v2_compare_sel_dow_tue.location_id =
            model1_v2_compare_sel_dow_wed.location_id AND
        model1_v2_compare_sel_dow_tue.stop_time =
            model1_v2_compare_sel_dow_wed.stop_time;

CREATE TABLE model1_v2_compare_project ENGINE = MEMORY
SELECT
    t1_route_number AS route_number,
    t1_location_id AS location_id,
    t1_stop_time AS stop_time,
    CAST(TRUNCATE(t1_avg_delay / 60, 0) AS SIGNED INTEGER) AS
        dow1_delay,
    CAST(TRUNCATE(t2_avg_delay / 60, 0) AS SIGNED INTEGER) AS
        dow2_delay
FROM model1_v2_compare_join;

CREATE TABLE model1_v2_compare ENGINE = MEMORY
SELECT *
FROM model1_v2_compare_project
ORDER BY location_id, stop_time;


-- STMT: 10
CREATE TABLE model2_v2_cleaned_base_data ENGINE = MEMORY
SELECT t1.*
FROM
    model1_v2_select_base_data_with_delay AS t1,
    (
        SELECT
```

```
289             service_date, dow_class,
290             route_number,
291             location_id, stop_time,
292             min(delay) AS min_delay
293         FROM model1_v2_select_base_data_with_delay
294         GROUP BY
295             service_date, dow_class,
296             route_number, location_id,
297             stop_time
298     ) AS t2
299 WHERE
300     t1.delay = t2.min_delay AND
301     t1.service_date = t2.service_date AND
302     t1.dow_class = t2.dow_class AND
303     t1.route_number = t2.route_number AND
304     t1.location_id = t2.location_id AND
305     t1.stop_time = t2.stop_time;
306
307 CREATE TABLE model2_v2_base_model ENGINE = MEMORY
308 SELECT
309     dow_class, route_number, location_id,
310     stop_time, STDDEV(delay) AS std_delay,
311     AVG(delay) AS avg_delay
312 FROM model2_v2_cleaned_base_data
313 GROUP BY
314     dow_class, route_number,
315     location_id, stop_time;
316
317
318 CREATE TABLE model2_v2_final_res_join ENGINE = MEMORY
319 SELECT
320     model2_v2_base_model.dow_class AS t2_dow_class,
```

```
    model2_v2_base_model.route_number AS t2_route_number,
    model2_v2_base_model.location_id AS t2_location_id,
    model2_v2_base_model.stop_time AS t2_stop_time,
    model2_v2_base_model.std_delay AS t2_std_delay,
    model2_v2_base_model.avg_delay AS t2_avg_delay,
    model2_v2_cleaned_base_data.service_date AS t1_service_date,
    model2_v2_cleaned_base_data.leave_time AS t1_leave_time,
    model2_v2_cleaned_base_data.route_number AS t1_route_number,
    model2_v2_cleaned_base_data.stop_time AS t1_stop_time,
    model2_v2_cleaned_base_data.arrive_time AS t1_arrive_time,
    model2_v2_cleaned_base_data.location_id AS t1_location_id,
    model2_v2_cleaned_base_data.schedule_status AS t1_schedule_status,
    model2_v2_cleaned_base_data.day_of_week AS t1_day_of_week,
    model2_v2_cleaned_base_data.dow_class AS t1_dow_class,
    model2_v2_cleaned_base_data.delay AS t1_delay
FROM
    model2_v2_base_model
    LEFT JOIN model2_v2_cleaned_base_data ON
        model2_v2_base_model.dow_class = model2_v2_cleaned_base_data.
            dow_class AND
        model2_v2_base_model.route_number = model2_v2_cleaned_base_data
            .route_number AND
        model2_v2_base_model.location_id = model2_v2_cleaned_base_data.
            location_id AND
        model2_v2_base_model.stop_time = model2_v2_cleaned_base_data.
            stop_time AND
        ABS(model2_v2_cleaned_base_data.delay) <= ABS(
            model2_v2_base_model.avg_delay) + model2_v2_base_model.
            std_delay;

CREATE TABLE model2_v2_final_res_agg ENGINE = MEMORY
SELECT
```

```
347     t2_dow_class, t2_route_number,
348     t2_location_id, t2_stop_time,
349     t2_avg_delay, AVG(t1_delay) AS delay
350 FROM model2_v2_final_res_join
351 GROUP BY
352     t2_dow_class, t2_route_number,
353     t2_location_id, t2_stop_time, t2_avg_delay;
354
355 CREATE TABLE model2_v2 ENGINE = MEMORY
356 SELECT
357     t2_dow_class AS dow_class,
358     t2_route_number AS route_number,
359     t2_location_id AS location_id,
360     t2_stop_time AS stop_time,
361     CAST(
362         TRUNCATE(COALESCE(delay, t2_avg_delay), 0)
363         AS SIGNED INTEGER
364     ) AS avg_delay
365 FROM model2_v2_final_res_agg;
366
367
368
369 -- STMT: 11
370 CREATE TABLE model2_v2_2_avg_delay_per_dow_class ENGINE = MEMORY
371 SELECT *
372 FROM stop_events_with_dow
373 WHERE
374     service_date >= '2018-11-01' AND
375     service_date < '2019-02-01';
376
377 CREATE TABLE model2_v2_2_agg ENGINE = MEMORY
378 SELECT
```

```
379     dow_class, route_number,
380     location_id, stop_time,
381     AVG(arrive_time - stop_time) AS avg_delay_raw,
382     count(*) AS num_of_observations
383 FROM model2_v2_2_avg_delay_per_dow_class
384 GROUP BY
385     dow_class, route_number,
386     location_id, stop_time;
387
388 CREATE TABLE model2_v2_2_proj ENGINE = MEMORY
389 SELECT
390     t1.*,
391     CAST(TRUNCATE(avg_delay_raw, 0) AS SIGNED INTEGER) AS avg_delay
392 FROM model2_v2_2_agg AS t1;
393
394
395 -- STMT: 12
396 CREATE TABLE compare_v2_m1_m2_sel_m1 ENGINE = MEMORY
397 SELECT *
398 FROM model1_v2
399 WHERE
400     day_of_week = 5 AND
401     route_number in (76, 78) AND
402     location_id = 2285;
403
404 CREATE TABLE compare_v2_m1_m2_sel_m2 ENGINE = MEMORY
405 SELECT *
406 FROM model2_v2
407 WHERE dow_class = 'D';
408
409 CREATE TABLE compare_v2_m1_m2_join ENGINE = MEMORY
410 SELECT
```

```
    compare_v2_m1_m2_sel_m1.day_of_week AS t1_day_of_week,
    compare_v2_m1_m2_sel_m1.route_number AS t1_route_number,
    compare_v2_m1_m2_sel_m1.location_id AS t1_location_id,
    compare_v2_m1_m2_sel_m1.stop_time AS t1_stop_time,
    compare_v2_m1_m2_sel_m1.avg_delay AS t1_avg_delay,
    compare_v2_m1_m2_sel_m2.dow_class AS t2_dow_class,
    compare_v2_m1_m2_sel_m2.route_number AS t2_route_number,
    compare_v2_m1_m2_sel_m2.location_id AS t2_location_id,
    compare_v2_m1_m2_sel_m2.stop_time AS t2_stop_time,
    compare_v2_m1_m2_sel_m2.avg_delay AS t2_avg_delay
FROM
    compare_v2_m1_m2_sel_m1
    JOIN compare_v2_m1_m2_sel_m2 ON
        compare_v2_m1_m2_sel_m1.route_number = compare_v2_m1_m2_sel_m2.
            route_number AND
        compare_v2_m1_m2_sel_m1.location_id = compare_v2_m1_m2_sel_m2.
            location_id AND
        compare_v2_m1_m2_sel_m1.stop_time = compare_v2_m1_m2_sel_m2.
            stop_time;

CREATE TABLE compare_v2_m1_m2_project ENGINE = MEMORY
SELECT
    t1_route_number AS route_number,
    t1_location_id AS location_id,
    t1_stop_time AS stop_time,
    CAST(TRUNCATE(t1_avg_delay / 60, 0) AS SIGNED INTEGER) AS
        dow1_delay,
    CAST(TRUNCATE(t2_avg_delay / 60, 0) AS SIGNED INTEGER) AS
        dow2_delay
FROM compare_v2_m1_m2_join;

CREATE TABLE compare_v2_m1_m2 ENGINE = MEMORY
```

```sql
SELECT *
FROM compare_v2_m1_m2_project
ORDER BY location_id, stop_time;


-- STMT: 13
CREATE TABLE baseline_l1 ENGINE = MEMORY
SELECT *
FROM stop_events_with_dow
WHERE
    service_date >= '2019-02-01' AND
    service_date < '2019-03-01';

CREATE TABLE baseline_l3 ENGINE = MEMORY
SELECT delay, COUNT(*) AS observations
FROM baseline_l1
GROUP BY delay;

CREATE TABLE baseline_l4 ENGINE = MEMORY
SELECT
    t1.*,
    CASE
        WHEN ABS(delay) > 5 THEN 'others'
        ELSE CAST(delay AS TEXT)
    END AS delay_diffs
FROM baseline_l3 AS t1;

CREATE TABLE baseline_l6 ENGINE = MEMORY
SELECT delay_diffs, SUM(observations) AS observations
FROM baseline_l4
GROUP BY delay_diffs;

```

```
CREATE TABLE baseline_l7 ENGINE = MEMORY
SELECT delay_diffs, observations
FROM baseline_l6;

CREATE TABLE baseline_l8 ENGINE = MEMORY
SELECT *
FROM baseline_l7
ORDER BY delay_diffs;


-- STMT: 14
CREATE TABLE baseline_rush_hour_l1 ENGINE = MEMORY
SELECT *
FROM baseline_l1
WHERE
    stop_time BETWEEN 23160 AND 31140 OR
    stop_time BETWEEN 57600 AND 66780;

CREATE TABLE baseline_rush_hour_l4 ENGINE = MEMORY
SELECT delay, COUNT(*) AS observations
FROM baseline_rush_hour_l1
GROUP BY delay;

CREATE TABLE baseline_rush_hour_l5 ENGINE = MEMORY
SELECT *
FROM baseline_rush_hour_l4
ORDER BY observations DESC;


-- STMT: 15
CREATE TABLE predicting_feb_arrival_l1 ENGINE = MEMORY
SELECT *
```

```
502 FROM baseline_l1
503 WHERE
504     route_number != 0 AND
505     schedule_status != 6;
506
507 CREATE TABLE predicting_feb_arrival_l2 ENGINE = MEMORY
508 SELECT
509     t1.*,
510     CAST(
511         TRUNCATE(CASE
512             WHEN ABS(arrive_time - stop_time) <= ABS(leave_time -
                    stop_time)
513                 THEN arrive_time - stop_time
514             ELSE leave_time - stop_time
515         END / 60, 0)
516         AS SIGNED INTEGER
517     ) AS actual_delay_in_min
518 FROM predicting_feb_arrival_l1 AS t1;
519
520 CREATE TABLE predicting_feb_arrival_l3 ENGINE = MEMORY
521 SELECT
522     predicting_feb_arrival_l2.service_date AS t1_service_date,
523     predicting_feb_arrival_l2.leave_time AS t1_leave_time,
524     predicting_feb_arrival_l2.route_number AS t1_route_number,
525     predicting_feb_arrival_l2.stop_time AS t1_stop_time,
526     predicting_feb_arrival_l2.arrive_time AS t1_arrive_time,
527     predicting_feb_arrival_l2.location_id AS t1_location_id,
528     predicting_feb_arrival_l2.schedule_status AS t1_schedule_status,
529     predicting_feb_arrival_l2.day_of_week AS t1_day_of_week,
530     predicting_feb_arrival_l2.delay AS t1_delay,
531     predicting_feb_arrival_l2.dow_class AS t1_dow_class,
```

```
532     predicting_feb_arrival_l2.actual_delay_in_min AS
            t1_actual_delay_in_min,
533     model1_v2.day_of_week AS t2_day_of_week,
534     model1_v2.route_number AS t2_route_number,
535     model1_v2.location_id AS t2_location_id,
536     model1_v2.stop_time AS t2_stop_time,
537     model1_v2.avg_delay AS t2_avg_delay
538 FROM predicting_feb_arrival_l2
539     JOIN model1_v2 ON
540         predicting_feb_arrival_l2.route_number = model1_v2.route_number
                AND
541         predicting_feb_arrival_l2.location_id = model1_v2.location_id
               AND
542         predicting_feb_arrival_l2.stop_time = model1_v2.stop_time AND
543         predicting_feb_arrival_l2.day_of_week = model1_v2.day_of_week;
544
545 CREATE TABLE predicting_feb_arrival_l4 ENGINE = MEMORY
546 SELECT
547     t1.*,
548     CAST(
549         TRUNCATE(t2_avg_delay / 60, 0)
550         AS SIGNED INTEGER
551     ) - t1_actual_delay_in_min AS delay_diff
552 FROM predicting_feb_arrival_l3 AS t1;
553
554 CREATE TABLE predicting_feb_arrival_l6 ENGINE = MEMORY
555 SELECT delay_diff, COUNT(*) AS observations
556 FROM predicting_feb_arrival_l4
557 GROUP BY delay_diff;
558
559 CREATE TABLE predicting_feb_arrival_l7 ENGINE = MEMORY
560 SELECT
```

```
561     t1.*,
562     CASE
563         WHEN ABS(delay_diff) > 3 THEN 'others'
564         ELSE CAST(delay_diff AS TEXT)
565     END AS delay_diffs
566 FROM predicting_feb_arrival_l6 AS t1;
567
568 CREATE TABLE predicting_feb_arrival_l10 ENGINE = MEMORY
569 SELECT delay_diffs, SUM(observations) AS observations
570 FROM predicting_feb_arrival_l7
571 GROUP BY delay_diffs;
572
573 CREATE TABLE predicting_feb_arrival_l11 ENGINE = MEMORY
574 SELECT *
575 FROM predicting_feb_arrival_l10
576 ORDER BY delay_diffs;
577
578
579 -- STMT: 16
580 CREATE TABLE predicting_feb_arrival_rush_hr_l1 ENGINE = MEMORY
581 SELECT *
582 FROM baseline_l1
583 WHERE
584     stop_time BETWEEN 23160 AND 31140 OR
585     stop_time BETWEEN 57600 AND 66780;
586
587 CREATE TABLE predicting_feb_arrival_rush_hr_l2 ENGINE = MEMORY
588 SELECT
589     t1.*,
590     CAST(TRUNCATE(delay / 60, 0) AS SIGNED INTEGER) AS
          actual_delay_in_min
591 FROM predicting_feb_arrival_rush_hr_l1 AS t1;
```

```
592
593 CREATE TABLE predicting_feb_arrival_rush_hr_l3 ENGINE = MEMORY
594 SELECT
595     predicting_feb_arrival_rush_hr_l2.service_date AS t1_service_date,
596     predicting_feb_arrival_rush_hr_l2.leave_time AS t1_leave_time,
597     predicting_feb_arrival_rush_hr_l2.route_number AS t1_route_number,
598     predicting_feb_arrival_rush_hr_l2.stop_time AS t1_stop_time,
599     predicting_feb_arrival_rush_hr_l2.arrive_time AS t1_arrive_time,
600     predicting_feb_arrival_rush_hr_l2.location_id AS t1_location_id,
601     predicting_feb_arrival_rush_hr_l2.schedule_status AS
            t1_schedule_status,
602     predicting_feb_arrival_rush_hr_l2.day_of_week AS t1_day_of_week,
603     predicting_feb_arrival_rush_hr_l2.delay AS t1_delay,
604     predicting_feb_arrival_rush_hr_l2.dow_class AS t1_dow_class,
605     predicting_feb_arrival_rush_hr_l2.actual_delay_in_min AS
            t1_actual_delay_in_min,
606     model1_v2.day_of_week AS t2_day_of_week,
607     model1_v2.route_number AS t2_route_number,
608     model1_v2.location_id AS t2_location_id,
609     model1_v2.stop_time AS t2_stop_time,
610     model1_v2.avg_delay AS t2_avg_delay
611 FROM predicting_feb_arrival_rush_hr_l2
612     JOIN model1_v2 ON
613         predicting_feb_arrival_rush_hr_l2.route_number = model1_v2.
                route_number AND
614         predicting_feb_arrival_rush_hr_l2.location_id = model1_v2.
                location_id AND
615         predicting_feb_arrival_rush_hr_l2.stop_time = model1_v2.
                stop_time AND
616         predicting_feb_arrival_rush_hr_l2.day_of_week = model1_v2.
                day_of_week;
617
```

```sql
CREATE TABLE predicting_feb_arrival_rush_hr_l4 ENGINE = MEMORY
SELECT
    t1.*,
    CAST(TRUNCATE(t2_avg_delay / 60, 0) AS SIGNED INTEGER) -
        t1_actual_delay_in_min AS delay_diff
FROM predicting_feb_arrival_rush_hr_l3 AS t1;

CREATE TABLE predicting_feb_arrival_rush_hr_l7 ENGINE = MEMORY
SELECT delay_diff, COUNT(*) AS observations
FROM predicting_feb_arrival_rush_hr_l4
GROUP BY delay_diff;

CREATE TABLE predicting_feb_arrival_rush_hr_l8 ENGINE = MEMORY
SELECT *
FROM predicting_feb_arrival_rush_hr_l7
ORDER BY observations DESC;


-- STMT: 17
CREATE TABLE predicting_feb_arrival_dow_class_l1 ENGINE = MEMORY
SELECT
    predicting_feb_arrival_l2.service_date AS t1_service_date,
    predicting_feb_arrival_l2.leave_time AS t1_leave_time,
    predicting_feb_arrival_l2.route_number AS t1_route_number,
    predicting_feb_arrival_l2.stop_time AS t1_stop_time,
    predicting_feb_arrival_l2.arrive_time AS t1_arrive_time,
    predicting_feb_arrival_l2.location_id AS t1_location_id,
    predicting_feb_arrival_l2.schedule_status AS t1_schedule_status,
    predicting_feb_arrival_l2.day_of_week AS t1_day_of_week,
    predicting_feb_arrival_l2.delay AS t1_delay,
    predicting_feb_arrival_l2.dow_class AS t1_dow_class,
```

```
648     predicting_feb_arrival_l2.actual_delay_in_min AS
            t1_actual_delay_in_min,
649     model2_v2_2_proj.dow_class AS t2_dow_class,
650     model2_v2_2_proj.route_number AS t2_route_number,
651     model2_v2_2_proj.location_id AS t2_location_id,
652     model2_v2_2_proj.stop_time AS t2_stop_time,
653     model2_v2_2_proj.avg_delay_raw AS t2_avg_delay_raw,
654     model2_v2_2_proj.num_of_observations AS t2_num_of_observations,
655     model2_v2_2_proj.avg_delay AS t2_avg_delay
656 FROM predicting_feb_arrival_l2
657     JOIN model2_v2_2_proj ON
658         predicting_feb_arrival_l2.route_number = model2_v2_2_proj.
                route_number AND
659         predicting_feb_arrival_l2.location_id = model2_v2_2_proj.
                location_id AND
660         predicting_feb_arrival_l2.stop_time = model2_v2_2_proj.
                stop_time AND
661         predicting_feb_arrival_l2.dow_class = model2_v2_2_proj.
                dow_class;
662
663 CREATE TABLE predicting_feb_arrival_dow_class_l2 ENGINE = MEMORY
664 SELECT
665     t1.*,
666     CAST(TRUNCATE(t2_avg_delay / 60, 0) AS SIGNED INTEGER) -
            t1_actual_delay_in_min AS delay_diff
667 FROM predicting_feb_arrival_dow_class_l1 AS t1;
668
669 CREATE TABLE predicting_feb_arrival_dow_class_l4 ENGINE = MEMORY
670 SELECT
671     delay_diff,
672     COUNT(*) AS observations
673 FROM predicting_feb_arrival_dow_class_l2
```

```sql
GROUP BY delay_diff;

CREATE TABLE predicting_feb_arrival_dow_class_l5 ENGINE = MEMORY
SELECT
    t1.*,
    CASE
        WHEN ABS(delay_diff) > 3 THEN 'others'
        ELSE CAST(delay_diff AS TEXT)
    END AS delay_diffs
FROM predicting_feb_arrival_dow_class_l4 AS t1;

CREATE TABLE predicting_feb_arrival_dow_class_l8 ENGINE = MEMORY
SELECT
    delay_diffs,
    SUM(observations) AS observations
FROM predicting_feb_arrival_dow_class_l5
GROUP BY delay_diffs;

CREATE TABLE predicting_feb_arrival_dow_class_l9 ENGINE = MEMORY
SELECT *
FROM predicting_feb_arrival_dow_class_l8
ORDER BY delay_diffs;


-- STMT: 18
CREATE TABLE predicting_feb_arrival_rush_hr_dow_class_l1 ENGINE =
    MEMORY
SELECT
    predicting_feb_arrival_rush_hr_l2.service_date AS t1_service_date,
    predicting_feb_arrival_rush_hr_l2.leave_time AS t1_leave_time,
    predicting_feb_arrival_rush_hr_l2.route_number AS t1_route_number,
    predicting_feb_arrival_rush_hr_l2.stop_time AS t1_stop_time,
```

```
705     predicting_feb_arrival_rush_hr_l2.arrive_time AS t1_arrive_time,
706     predicting_feb_arrival_rush_hr_l2.location_id AS t1_location_id,
707     predicting_feb_arrival_rush_hr_l2.schedule_status AS
            t1_schedule_status,
708     predicting_feb_arrival_rush_hr_l2.day_of_week AS t1_day_of_week,
709     predicting_feb_arrival_rush_hr_l2.delay AS t1_delay,
710     predicting_feb_arrival_rush_hr_l2.dow_class AS t1_dow_class,
711     predicting_feb_arrival_rush_hr_l2.actual_delay_in_min AS
            t1_actual_delay_in_min,
712     model2_v2_2_proj.dow_class AS t2_dow_class,
713     model2_v2_2_proj.route_number AS t2_route_number,
714     model2_v2_2_proj.location_id AS t2_location_id,
715     model2_v2_2_proj.stop_time AS t2_stop_time,
716     model2_v2_2_proj.avg_delay_raw AS t2_avg_delay_raw,
717     model2_v2_2_proj.num_of_observations AS t2_num_of_observations,
718     model2_v2_2_proj.avg_delay AS t2_avg_delay
719 FROM predicting_feb_arrival_rush_hr_l2
720     JOIN model2_v2_2_proj ON
721         predicting_feb_arrival_rush_hr_l2.route_number =
                model2_v2_2_proj.route_number AND
722         predicting_feb_arrival_rush_hr_l2.location_id =
                model2_v2_2_proj.location_id AND
723         predicting_feb_arrival_rush_hr_l2.stop_time = model2_v2_2_proj.
                stop_time AND
724         predicting_feb_arrival_rush_hr_l2.dow_class = model2_v2_2_proj.
                dow_class;
725
726 CREATE TABLE predicting_feb_arrival_rush_hr_dow_class_l2 ENGINE =
        MEMORY
727 SELECT
728     t1.*,
```

```
    CAST(TRUNCATE(t2_avg_delay / 60, 0) AS SIGNED INTEGER) -
        t1_actual_delay_in_min AS delay_diff
FROM predicting_feb_arrival_rush_hr_dow_class_l1 AS t1;

CREATE TABLE predicting_feb_arrival_rush_hr_dow_class_l5 ENGINE =
    MEMORY
SELECT
    delay_diff,
    COUNT(*) AS observations
FROM predicting_feb_arrival_rush_hr_dow_class_l2
GROUP BY delay_diff;

CREATE TABLE predicting_feb_arrival_rush_hr_dow_class_l6 ENGINE =
    MEMORY
SELECT *
FROM predicting_feb_arrival_rush_hr_dow_class_l5
ORDER BY observations DESC;


-- STMT: 19
CREATE TABLE model1_v3_l2 ENGINE = MEMORY
SELECT
    service_date, route_number,
    location_id, stop_time,
    MAX(arrive_time) AS arrive_time,
    MAX(leave_time) AS leave_time
FROM model1_v2_select_base_data
GROUP BY
    service_date, route_number,
    location_id, stop_time;

CREATE TABLE model1_v3_l3 ENGINE = MEMORY
```

```
SELECT
    t1.*,
    DAYOFWEEK(service_date) - 1 AS day_of_week
FROM model1_v3_l2 AS t1;

CREATE TABLE model1_v3_l5 ENGINE = MEMORY
SELECT
    day_of_week, route_number,
    location_id, stop_time,
    STDDEV(arrive_time) AS std_arrive_time,
    AVG(arrive_time) AS avg_arrive_time,
    STDDEV(leave_time) AS std_leave_time,
    AVG(leave_time) AS avg_leave_time
FROM model1_v3_l3
GROUP BY
    day_of_week, route_number,
    location_id, stop_time;

CREATE TABLE model1_v3_l6 ENGINE = MEMORY
SELECT
    model1_v3_l3.service_date AS t1_service_date,
    model1_v3_l3.route_number AS t1_route_number,
    model1_v3_l3.location_id AS t1_location_id,
    model1_v3_l3.stop_time AS t1_stop_time,
    model1_v3_l3.arrive_time AS t1_arrive_time,
    model1_v3_l3.leave_time AS t1_leave_time,
    model1_v3_l3.day_of_week AS t1_day_of_week,
    model1_v3_l5.day_of_week AS t2_day_of_week,
    model1_v3_l5.route_number AS t2_route_number,
    model1_v3_l5.location_id AS t2_location_id,
    model1_v3_l5.stop_time AS t2_stop_time,
    model1_v3_l5.std_arrive_time AS t2_std_arrive_time,
```

```
    model1_v3_l5.avg_arrive_time AS t2_avg_arrive_time,
    model1_v3_l5.std_leave_time AS t2_std_leave_time,
    model1_v3_l5.avg_leave_time AS t2_avg_leave_time
FROM model1_v3_l3
    JOIN model1_v3_l5 ON
        model1_v3_l5.day_of_week = model1_v3_l3.day_of_week AND
        model1_v3_l5.route_number = model1_v3_l3.route_number AND
        model1_v3_l5.location_id = model1_v3_l3.location_id AND
        model1_v3_l5.stop_time = model1_v3_l3.stop_time;

CREATE TABLE model1_v3_l8 ENGINE = MEMORY
SELECT
    t1.t2_day_of_week, t1.t2_route_number,
    t1.t2_location_id, t1.t2_stop_time,
    t1.t2_avg_arrive_time, t1.t2_avg_leave_time,
    t2.avg_arrive_time, t2.avg_leave_time
FROM
    (
        SELECT DISTINCT
            t2_day_of_week, t2_route_number,
            t2_location_id, t2_stop_time,
            t2_avg_arrive_time, t2_avg_leave_time
        FROM model1_v3_l6
    ) AS t1
    LEFT JOIN (
        SELECT
            IFNULL(t3.t2_day_of_week, t4.t2_day_of_week) AS
                t2_day_of_week,
            IFNULL(t3.t2_route_number, t4.t2_route_number) AS
                t2_route_number,
            IFNULL(t3.t2_location_id, t4.t2_location_id) AS
                t2_location_id,
```

```
819             IFNULL(t3.t2_stop_time, t4.t2_stop_time) AS t2_stop_time,
820             IFNULL(t3.t2_avg_arrive_time, t4.t2_avg_arrive_time) AS
                    t2_avg_arrive_time,
821             IFNULL(t3.t2_avg_leave_time, t4.t2_avg_leave_time) AS
                    t2_avg_leave_time,
822             t3.avg_arrive_time,
823             t4.avg_leave_time
824         FROM (
825             SELECT
826                 t2_day_of_week, t2_route_number,
827                 t2_location_id, t2_stop_time,
828                 t2_avg_arrive_time, t2_avg_leave_time,
829                 AVG(t1_arrive_time) AS avg_arrive_time
830             FROM model1_v3_l6
831             WHERE
832                 abs(t1_arrive_time) <= abs(t2_avg_arrive_time) +
                        t2_std_arrive_time
833             GROUP BY
834                 t2_day_of_week, t2_route_number,
835                 t2_location_id, t2_stop_time,
836                 t2_avg_arrive_time, t2_avg_leave_time
837         ) AS t3
838         FULL JOIN (
839             SELECT
840                 t2_day_of_week, t2_route_number,
841                 t2_location_id, t2_stop_time,
842                 t2_avg_arrive_time, t2_avg_leave_time,
843                 AVG(t1_leave_time) AS avg_leave_time
844             FROM model1_v3_l6
845             WHERE
846                 abs(t1_leave_time) <= abs(t2_avg_leave_time) +
                        t2_std_leave_time
```

```
            GROUP BY
                t2_day_of_week, t2_route_number,
                t2_location_id, t2_stop_time,
                t2_avg_arrive_time, t2_avg_leave_time
        ) AS t4 ON
            t3.t2_day_of_week = t4.t2_day_of_week AND
            t3.t2_route_number = t4.t2_route_number AND
            t3.t2_location_id = t4.t2_location_id AND
            t3.t2_stop_time = t4.t2_stop_time AND
            t3.t2_avg_arrive_time = t4.t2_avg_arrive_time AND
            t3.t2_avg_leave_time = t4.t2_avg_leave_time
    ) AS t2 ON
        t1.t2_day_of_week = t2.t2_day_of_week AND
        t1.t2_route_number = t2.t2_route_number AND
        t1.t2_location_id = t2.t2_location_id AND
        t1.t2_stop_time = t2.t2_stop_time AND
        t1.t2_avg_arrive_time = t2.t2_avg_arrive_time AND
        t1.t2_avg_leave_time = t2.t2_avg_leave_time;


CREATE TABLE model1_v3_l9 ENGINE = MEMORY
SELECT
    t2_day_of_week AS day_of_week,
    t2_route_number AS route_number,
    t2_location_id AS location_id,
    t2_stop_time AS stop_time,
    CAST(
        TRUNCATE(COALESCE(avg_arrive_time, t2_avg_arrive_time), 0)
        AS SIGNED INTEGER
    ) AS arrive_time,
    CAST(
        TRUNCATE(COALESCE(avg_leave_time, t2_avg_leave_time), 0)
```

```
        AS SIGNED INTEGER
    ) AS leave_time
FROM model1_v3_l8;


-- STMT: 20
CREATE TABLE model2_v3_l1 ENGINE = MEMORY
SELECT t1.*,
    CASE
        WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
        WHEN day_of_week = 0 THEN 'U'
        ELSE 'S'
    END AS dow_class
FROM model1_v3_l3 AS t1;

CREATE TABLE model2_v3_l3 ENGINE = MEMORY
SELECT
    dow_class, route_number,
    location_id, stop_time,
    STDDEV(arrive_time) AS std_arrive_time,
    AVG(arrive_time) AS avg_arrive_time,
    STDDEV(leave_time) AS std_leave_time,
    AVG(leave_time) AS avg_leave_time
FROM model2_v3_l1
GROUP BY
    dow_class, route_number,
    location_id, stop_time;

CREATE TABLE model2_v3_l4 ENGINE = MEMORY
SELECT
    model2_v3_l1.service_date AS t1_service_date,
    model2_v3_l1.route_number AS t1_route_number,
```

```
    model2_v3_l1.location_id AS t1_location_id,
    model2_v3_l1.stop_time AS t1_stop_time,
    model2_v3_l1.arrive_time AS t1_arrive_time,
    model2_v3_l1.leave_time AS t1_leave_time,
    model2_v3_l1.day_of_week AS t1_day_of_week,
    model2_v3_l1.dow_class AS t1_dow_class,
    model2_v3_l3.dow_class AS t2_dow_class,
    model2_v3_l3.route_number AS t2_route_number,
    model2_v3_l3.location_id AS t2_location_id,
    model2_v3_l3.stop_time AS t2_stop_time,
    model2_v3_l3.std_arrive_time AS t2_std_arrive_time,
    model2_v3_l3.avg_arrive_time AS t2_avg_arrive_time,
    model2_v3_l3.std_leave_time AS t2_std_leave_time,
    model2_v3_l3.avg_leave_time AS t2_avg_leave_time
FROM model2_v3_l1
    JOIN model2_v3_l3 ON
        model2_v3_l1.dow_class = model2_v3_l3.dow_class AND
        model2_v3_l1.route_number = model2_v3_l3.route_number AND
        model2_v3_l1.location_id = model2_v3_l3.location_id AND
        model2_v3_l1.stop_time = model2_v3_l3.stop_time;


CREATE TABLE model2_v3_l6 ENGINE = MEMORY
SELECT
    t1.t2_dow_class, t1.t2_route_number,
    t1.t2_location_id, t1.t2_stop_time,
    t1.t2_avg_arrive_time, t1.t2_avg_leave_time,
    t2.avg_arrive_time, t2.avg_leave_time
FROM
    (
        SELECT DISTINCT
            t2_dow_class, t2_route_number,
```

```
            t2_location_id, t2_stop_time,
            t2_avg_arrive_time, t2_avg_leave_time
        FROM model2_v3_l4
    ) AS t1
    LEFT JOIN (
        SELECT
            IFNULL(t3.t2_dow_class, t4.t2_dow_class) AS t2_dow_class,
            IFNULL(t3.t2_route_number, t4.t2_route_number) AS
                t2_route_number,
            IFNULL(t3.t2_location_id, t4.t2_location_id) AS
                t2_location_id,
            IFNULL(t3.t2_stop_time, t4.t2_stop_time) AS t2_stop_time,
            IFNULL(t3.t2_avg_arrive_time, t4.t2_avg_arrive_time) AS
                t2_avg_arrive_time,
            IFNULL(t3.t2_avg_leave_time, t4.t2_avg_leave_time) AS
                t2_avg_leave_time,
            t3.avg_arrive_time,
            t4.avg_leave_time
        FROM
            (
                SELECT
                    t2_dow_class, t2_route_number,
                    t2_location_id, t2_stop_time,
                    t2_avg_arrive_time, t2_avg_leave_time,
                    AVG(t1_arrive_time) AS avg_arrive_time
                FROM model2_v3_l4
                WHERE
                    abs(t1_arrive_time) <= abs(t2_avg_arrive_time) +
                        t2_std_arrive_time
                GROUP BY
                    t2_dow_class, t2_route_number,
                    t2_location_id, t2_stop_time,
```

```
970                     t2_avg_arrive_time, t2_avg_leave_time
971             ) AS t3
972             FULL JOIN (
973                 SELECT
974                     t2_dow_class, t2_route_number,
975                     t2_location_id, t2_stop_time,
976                     t2_avg_arrive_time, t2_avg_leave_time,
977                     AVG(t1_leave_time) AS avg_leave_time
978                 FROM model2_v3_l4
979                 WHERE
980                     abs(t1_leave_time) <= abs(t2_avg_leave_time) +
                        t2_std_leave_time
981                 GROUP BY
982                     t2_dow_class, t2_route_number,
983                     t2_location_id, t2_stop_time,
984                     t2_avg_arrive_time, t2_avg_leave_time
985             ) AS t4 ON
986                 t3.t2_dow_class = t4.t2_dow_class AND
987                 t3.t2_route_number = t4.t2_route_number AND
988                 t3.t2_location_id = t4.t2_location_id AND
989                 t3.t2_stop_time = t4.t2_stop_time AND
990                 t3.t2_avg_arrive_time = t4.t2_avg_arrive_time AND
991                 t3.t2_avg_leave_time = t4.t2_avg_leave_time
992     ) AS t2 ON
993         t1.t2_dow_class = t2.t2_dow_class AND
994         t1.t2_route_number = t2.t2_route_number AND
995         t1.t2_location_id = t2.t2_location_id AND
996         t1.t2_stop_time = t2.t2_stop_time AND
997         t1.t2_avg_arrive_time = t2.t2_avg_arrive_time AND
998         t1.t2_avg_leave_time = t2.t2_avg_leave_time;
999
1000
```

```
CREATE TABLE model2_v3_l7 ENGINE = MEMORY
SELECT
    t2_dow_class AS dow_class,
    t2_route_number AS route_number,
    t2_location_id AS location_id,
    t2_stop_time AS stop_time,
    CAST(
        TRUNCATE(COALESCE(avg_arrive_time, t2_avg_arrive_time), 0)
        AS SIGNED INTEGER
    ) AS arrive_time,
    CAST(
        TRUNCATE(COALESCE(avg_leave_time, t2_avg_leave_time), 0)
        AS SIGNED INTEGER
    ) AS leave_time
FROM model2_v3_l6;


-- STMT: 21
CREATE TABLE baseline_v2_l1 ENGINE = MEMORY
SELECT *
FROM baseline_l1
WHERE route_number <> 0;

CREATE TABLE baseline_v2_l3 ENGINE = MEMORY
SELECT
    service_date, route_number,
    location_id, stop_time,
    MIN(arrive_time) AS arrive_time,
    MAX(leave_time) AS leave_time
FROM baseline_v2_l1
GROUP BY
    service_date, route_number,
```

```
    location_id, stop_time;

CREATE TABLE baseline_v2_l4 ENGINE = MEMORY
SELECT
    CAST(
        TRUNCATE(CASE
            WHEN ABS(arrive_time - stop_time) <= ABS(leave_time - 30 -
                 stop_time)
                THEN arrive_time - stop_time
            ELSE leave_time - 30 - stop_time
        END / 60, 0)
        AS SIGNED INTEGER
    ) AS prediction_diff
FROM baseline_v2_l3;

CREATE TABLE baseline_v2_l5 ENGINE = MEMORY
SELECT
    CASE
        WHEN prediction_diff > 3 THEN 'others'
        ELSE CAST(prediction_diff AS TEXT)
    END AS prediction_diffs
FROM baseline_v2_l4;

CREATE TABLE baseline_v2_l8 ENGINE = MEMORY
SELECT
    prediction_diffs,
    COUNT(*) AS observations
FROM baseline_v2_l5
GROUP BY prediction_diffs;

CREATE TABLE baseline_v2_l9 ENGINE = MEMORY
SELECT *
```

```
FROM baseline_v2_l8
ORDER BY prediction_diffs;


-- STMT: 22
CREATE TABLE baseline_v2_rush_hour_l1 ENGINE = MEMORY
SELECT *
FROM baseline_rush_hour_l1
WHERE route_number <> 0;

CREATE TABLE baseline_v2_rush_hour_l3 ENGINE = MEMORY
SELECT
    service_date, route_number,
    location_id, stop_time,
    MIN(arrive_time) AS arrive_time,
    MAX(leave_time) AS leave_time
FROM baseline_v2_rush_hour_l1
GROUP BY
    service_date, route_number,
    location_id, stop_time;

CREATE TABLE baseline_v2_rush_hour_l4 ENGINE = MEMORY
SELECT
    CAST(
        TRUNCATE(CASE
            WHEN ABS(arrive_time - stop_time) <= ABS(leave_time - 30 -
                 stop_time)
                THEN arrive_time - stop_time
            ELSE leave_time - 30 - stop_time
        END / 60, 0)
        AS SIGNED INTEGER
    ) AS prediction_diff
```

```
FROM baseline_v2_rush_hour_l3;

CREATE TABLE baseline_v2_rush_hour_l5 ENGINE = MEMORY
SELECT
    CASE
        WHEN prediction_diff > 3 THEN 'others'
        ELSE CAST(prediction_diff AS TEXT)
    END AS prediction_diffs
FROM baseline_v2_rush_hour_l4;

CREATE TABLE baseline_v2_rush_hour_l8 ENGINE = MEMORY
SELECT
    prediction_diffs,
    COUNT(*) AS observations
FROM baseline_v2_rush_hour_l5
GROUP BY prediction_diffs;

CREATE TABLE baseline_v2_rush_hour_l9 ENGINE = MEMORY
SELECT *
FROM baseline_v2_rush_hour_l8
ORDER BY prediction_diffs;


-- STMT: 23
CREATE TABLE comp_predic_v2_l1 ENGINE = MEMORY
SELECT
    t1.*,
    DAYOFWEEK(service_date) - 1 AS day_of_week
FROM baseline_v2_l3 AS t1;

CREATE TABLE comp_predic_v2_l2 ENGINE = MEMORY
SELECT
```

```
    comp_predic_v2_l1.service_date AS t1_service_date,
    comp_predic_v2_l1.route_number AS t1_route_number,
    comp_predic_v2_l1.location_id AS t1_location_id,
    comp_predic_v2_l1.stop_time AS t1_stop_time,
    comp_predic_v2_l1.arrive_time AS t1_arrive_time,
    comp_predic_v2_l1.leave_time AS t1_leave_time,
    comp_predic_v2_l1.day_of_week AS t1_day_of_week,
    model1_v3_l9.day_of_week AS t2_day_of_week,
    model1_v3_l9.route_number AS t2_route_number,
    model1_v3_l9.location_id AS t2_location_id,
    model1_v3_l9.stop_time AS t2_stop_time,
    model1_v3_l9.arrive_time AS t2_arrive_time,
    model1_v3_l9.leave_time AS t2_leave_time
FROM comp_predic_v2_l1
    JOIN model1_v3_l9 ON
        comp_predic_v2_l1.route_number = model1_v3_l9.route_number AND
        comp_predic_v2_l1.location_id = model1_v3_l9.location_id AND
        comp_predic_v2_l1.stop_time = model1_v3_l9.stop_time AND
        comp_predic_v2_l1.day_of_week = model1_v3_l9.day_of_week;

CREATE TABLE comp_predic_v2_l3 ENGINE = MEMORY
SELECT
    t1.*,
    CAST(
        TRUNCATE((t2_arrive_time - t1_arrive_time) / 60, 0)
        AS SIGNED INTEGER
    ) AS prediction_diff
FROM comp_predic_v2_l2 AS t1;

CREATE TABLE comp_predic_v2_l5 ENGINE = MEMORY
SELECT
    prediction_diff,
```

```sql
    COUNT(*) AS observations
FROM comp_predic_v2_l3
GROUP BY prediction_diff;

CREATE TABLE comp_predic_v2_l6 ENGINE = MEMORY
SELECT
    t1.*,
    CASE
        WHEN prediction_diff > 3 THEN 'others'
        ELSE CAST(prediction_diff AS TEXT)
    END AS prediction_diffs
FROM comp_predic_v2_l5 AS t1;

CREATE TABLE comp_predic_v2_l9 ENGINE = MEMORY
SELECT
    prediction_diffs,
    SUM(observations) AS observations
FROM comp_predic_v2_l6
GROUP BY prediction_diffs;

CREATE TABLE comp_predic_v2_l10 ENGINE = MEMORY
SELECT *
FROM comp_predic_v2_l9
ORDER BY prediction_diffs;


-- STMT: 24
CREATE TABLE comp_predic_v2_rush_hour_l1 ENGINE = MEMORY
SELECT
    t1.*,
    DAYOFWEEK(service_date) - 1 AS day_of_week
FROM baseline_v2_rush_hour_l3 AS t1;
```

```
1191
1192 CREATE TABLE comp_predic_v2_rush_hour_l2 ENGINE = MEMORY
1193 SELECT
1194     comp_predic_v2_rush_hour_l1.service_date AS t1_service_date,
1195     comp_predic_v2_rush_hour_l1.route_number AS t1_route_number,
1196     comp_predic_v2_rush_hour_l1.location_id AS t1_location_id,
1197     comp_predic_v2_rush_hour_l1.stop_time AS t1_stop_time,
1198     comp_predic_v2_rush_hour_l1.arrive_time AS t1_arrive_time,
1199     comp_predic_v2_rush_hour_l1.leave_time AS t1_leave_time,
1200     comp_predic_v2_rush_hour_l1.day_of_week AS t1_day_of_week,
1201     model1_v3_l9.day_of_week AS t2_day_of_week,
1202     model1_v3_l9.route_number AS t2_route_number,
1203     model1_v3_l9.location_id AS t2_location_id,
1204     model1_v3_l9.stop_time AS t2_stop_time,
1205     model1_v3_l9.arrive_time AS t2_arrive_time,
1206     model1_v3_l9.leave_time AS t2_leave_time
1207 FROM comp_predic_v2_rush_hour_l1
1208     JOIN model1_v3_l9 ON
1209         comp_predic_v2_rush_hour_l1.route_number = model1_v3_l9.
                route_number AND
1210         comp_predic_v2_rush_hour_l1.location_id = model1_v3_l9.
                location_id AND
1211         comp_predic_v2_rush_hour_l1.stop_time = model1_v3_l9.stop_time
                AND
1212         comp_predic_v2_rush_hour_l1.day_of_week = model1_v3_l9.
                day_of_week;
1213
1214 CREATE TABLE comp_predic_v2_rush_hour_l3 ENGINE = MEMORY
1215 SELECT
1216     t1.*,
1217     CAST(
1218         TRUNCATE((t2_arrive_time - t1_arrive_time) / 60, 0)
```

```
        AS SIGNED INTEGER
    ) AS prediction_diff
FROM comp_predic_v2_rush_hour_l2 AS t1;

CREATE TABLE comp_predic_v2_rush_hour_l5 ENGINE = MEMORY
SELECT
    prediction_diff,
    COUNT(*) AS observations
FROM comp_predic_v2_rush_hour_l3
GROUP BY prediction_diff;

CREATE TABLE comp_predic_v2_rush_hour_l6 ENGINE = MEMORY
SELECT
    t1.*,
    CASE
        WHEN prediction_diff > 3 THEN 'others'
        ELSE CAST(prediction_diff AS TEXT)
    END AS prediction_diffs
FROM comp_predic_v2_rush_hour_l5 AS t1;

CREATE TABLE comp_predic_v2_rush_hour_l9 ENGINE = MEMORY
SELECT
    prediction_diffs,
    SUM(observations) AS observations
FROM comp_predic_v2_rush_hour_l6
GROUP BY prediction_diffs;

CREATE TABLE comp_predic_v2_rush_hour_l10 ENGINE = MEMORY
SELECT *
FROM comp_predic_v2_rush_hour_l9
ORDER BY prediction_diffs;

```

```
1251
1252 -- STMT: 25
1253 CREATE TABLE comp_predic_v3_l1 ENGINE = MEMORY
1254 SELECT
1255     t1.*,
1256     CASE
1257         WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
1258         WHEN day_of_week = 0 THEN 'U'
1259         ELSE 'S'
1260     END AS dow_class
1261 FROM comp_predic_v2_l1 AS t1;
1262
1263 CREATE TABLE comp_predic_v3_l2 ENGINE = MEMORY
1264 SELECT
1265     comp_predic_v3_l1.service_date AS t1_service_date,
1266     comp_predic_v3_l1.route_number AS t1_route_number,
1267     comp_predic_v3_l1.location_id AS t1_location_id,
1268     comp_predic_v3_l1.stop_time AS t1_stop_time,
1269     comp_predic_v3_l1.arrive_time AS t1_arrive_time,
1270     comp_predic_v3_l1.leave_time AS t1_leave_time,
1271     comp_predic_v3_l1.day_of_week AS t1_day_of_week,
1272     comp_predic_v3_l1.dow_class AS t1_dow_class,
1273     model2_v3_l7.dow_class AS t2_dow_class,
1274     model2_v3_l7.route_number AS t2_route_number,
1275     model2_v3_l7.location_id AS t2_location_id,
1276     model2_v3_l7.stop_time AS t2_stop_time,
1277     model2_v3_l7.arrive_time AS t2_arrive_time,
1278     model2_v3_l7.leave_time AS t2_leave_time
1279 FROM comp_predic_v3_l1
1280     JOIN model2_v3_l7 ON
1281         comp_predic_v3_l1.route_number = model2_v3_l7.route_number AND
1282         comp_predic_v3_l1.location_id = model2_v3_l7.location_id AND
```

```
        comp_predic_v3_l1.stop_time = model2_v3_l7.stop_time AND
        comp_predic_v3_l1.dow_class = model2_v3_l7.dow_class;

CREATE TABLE comp_predic_v3_l3 ENGINE = MEMORY
SELECT
    t1.*,
    CAST(
        TRUNCATE((t2_arrive_time - t1_arrive_time) / 60, 0)
        AS SIGNED INTEGER
    ) AS prediction_diff
FROM comp_predic_v3_l2 AS t1;

CREATE TABLE comp_predic_v3_l5 ENGINE = MEMORY
SELECT
    prediction_diff,
    COUNT(*) AS observations
FROM comp_predic_v3_l3
GROUP BY prediction_diff;

CREATE TABLE comp_predic_v3_l6 ENGINE = MEMORY
SELECT
    t1.*,
    CASE
        WHEN prediction_diff > 3 THEN 'others'
        ELSE CAST(prediction_diff AS TEXT)
    END AS prediction_diffs
FROM comp_predic_v3_l5 AS t1;

CREATE TABLE comp_predic_v3_l9 ENGINE = MEMORY
SELECT
    prediction_diffs,
    SUM(observations) AS observations
```

```
FROM comp_predic_v3_l6
GROUP BY prediction_diffs;

CREATE TABLE comp_predic_v3_l10 ENGINE = MEMORY
SELECT *
FROM comp_predic_v3_l9
ORDER BY prediction_diffs;


-- STMT: 26
CREATE TABLE comp_predic_v3_rush_hour_l1 ENGINE = MEMORY
SELECT
    t1.*,
    CASE
        WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
        WHEN day_of_week = 0 THEN 'U'
        ELSE 'S'
    END AS dow_class
FROM comp_predic_v2_rush_hour_l1 AS t1;

CREATE TABLE comp_predic_v3_rush_hour_l2 ENGINE = MEMORY
SELECT
    comp_predic_v3_rush_hour_l1.service_date AS t1_service_date,
    comp_predic_v3_rush_hour_l1.route_number AS t1_route_number,
    comp_predic_v3_rush_hour_l1.location_id AS t1_location_id,
    comp_predic_v3_rush_hour_l1.stop_time AS t1_stop_time,
    comp_predic_v3_rush_hour_l1.arrive_time AS t1_arrive_time,
    comp_predic_v3_rush_hour_l1.leave_time AS t1_leave_time,
    comp_predic_v3_rush_hour_l1.day_of_week AS t1_day_of_week,
    comp_predic_v3_rush_hour_l1.dow_class AS t1_dow_class,
    model2_v3_l7.dow_class AS t2_dow_class,
    model2_v3_l7.route_number AS t2_route_number,
```

```
    model2_v3_l7.location_id AS t2_location_id,
    model2_v3_l7.stop_time AS t2_stop_time,
    model2_v3_l7.arrive_time AS t2_arrive_time,
    model2_v3_l7.leave_time AS t2_leave_time
FROM comp_predic_v3_rush_hour_l1
    JOIN model2_v3_l7 ON
        comp_predic_v3_rush_hour_l1.route_number = model2_v3_l7.
            route_number AND
        comp_predic_v3_rush_hour_l1.location_id = model2_v3_l7.
            location_id AND
        comp_predic_v3_rush_hour_l1.stop_time = model2_v3_l7.stop_time
            AND
        comp_predic_v3_rush_hour_l1.dow_class = model2_v3_l7.dow_class;

CREATE TABLE comp_predic_v3_rush_hour_l3 ENGINE = MEMORY
SELECT
    t1.*,
    CAST(
        TRUNCATE((t2_arrive_time - t1_arrive_time) / 60, 0)
        AS SIGNED INTEGER
    ) AS prediction_diff
FROM comp_predic_v3_rush_hour_l2 AS t1;

CREATE TABLE comp_predic_v3_rush_hour_l5 ENGINE = MEMORY
SELECT
    prediction_diff,
    COUNT(*) AS observations
FROM comp_predic_v3_rush_hour_l3
GROUP BY prediction_diff;

CREATE TABLE comp_predic_v3_rush_hour_l6 ENGINE = MEMORY
SELECT
```

```
    t1.*,
    CASE
        WHEN prediction_diff > 3 THEN 'others'
        ELSE CAST(prediction_diff AS TEXT)
    END AS prediction_diffs
FROM comp_predic_v3_rush_hour_l5 AS t1;

CREATE TABLE comp_predic_v3_rush_hour_l9 ENGINE = MEMORY
SELECT
    prediction_diffs,
    SUM(observations) AS observations
FROM comp_predic_v3_rush_hour_l6
GROUP BY prediction_diffs;

CREATE TABLE comp_predic_v3_rush_hour_l10 ENGINE = MEMORY
SELECT *
FROM comp_predic_v3_rush_hour_l9
ORDER BY prediction_diffs;


-- STMT: 27
CREATE TABLE comp_pred_model1_and_model2_l1 ENGINE = MEMORY
SELECT
    model1_v3_l9.day_of_week AS t1_day_of_week,
    model1_v3_l9.route_number AS t1_route_number,
    model1_v3_l9.location_id AS t1_location_id,
    model1_v3_l9.stop_time AS t1_stop_time,
    model1_v3_l9.arrive_time AS t1_arrive_time,
    model1_v3_l9.leave_time AS t1_leave_time,
    model2_v3_l7.dow_class AS t2_dow_class,
    model2_v3_l7.route_number AS t2_route_number,
    model2_v3_l7.location_id AS t2_location_id,
```

```
    model2_v3_l7.stop_time AS t2_stop_time,
    model2_v3_l7.arrive_time AS t2_arrive_time,
    model2_v3_l7.leave_time AS t2_leave_time
FROM model1_v3_l9
    JOIN model2_v3_l7 ON
        model1_v3_l9.route_number = model2_v3_l7.route_number AND
        model1_v3_l9.location_id = model2_v3_l7.location_id AND
        model1_v3_l9.stop_time = model2_v3_l7.stop_time;

CREATE TABLE comp_pred_model1_and_model2_l2 ENGINE = MEMORY
SELECT *
FROM comp_pred_model1_and_model2_l1
WHERE
    t1_day_of_week = 5 AND
    t2_dow_class = 'D' AND
    t1_route_number in (76, 78) AND
    t1_location_id = 2285;

CREATE TABLE comp_pred_model1_and_model2_l3 ENGINE = MEMORY
SELECT
    t1_route_number AS route_number,
    t1_location_id AS location_id,
    t1_stop_time AS stop_time,
    t1_arrive_time AS model1_pred_arrival_time,
    t1_leave_time AS model1_pred_leave_time,
    t2_arrive_time AS model2_pred_arrival_time,
    t2_leave_time AS model2_pred_leave_time
FROM comp_pred_model1_and_model2_l2;

CREATE TABLE comp_pred_model1_and_model2_l4 ENGINE = MEMORY
SELECT *
FROM comp_pred_model1_and_model2_l3
```

```
1440 ORDER BY location_id, stop_time;
```

A.5.1 Min-Max Queries

The following are the min-max queries that we used to test data-access time[3] at the top of each of the 27 stacks in addition to stack 0 (the original data set).

```
1  -- MIN/MAX QUERIES FOR EVERY STACK
2
3  -- STACK 0: min_max_query0
4  SELECT
5      MIN(service_date) AS min_date,
6      MAX(service_date) AS max_date
7  FROM stop_events;
8
9  -- STACK 1: min_max_query1
10 SELECT
11     MIN(service_date) AS min_date,
12     MAX(service_date) AS max_date
13 FROM route58_stop910_ordered;
14
15 -- STACK 2: min_max_query2
16 SELECT
17     MIN(route_number) AS min_route_num,
18     MAX(route_number) AS max_route_num
19 FROM distinct_routes_at_stop9821;
20
21 -- STACK 3: min_max_query3
22 SELECT
23     MIN(service_date) AS min_date,
24     MAX(service_date) AS max_date
25 FROM duplicates;
```

[3]By using these queries, we are actually not measuring access time, but rather build time.

```sql

-- STACK 4: min_max_query4
SELECT
    MIN(service_date) AS min_date,
    MAX(service_date) AS max_date
FROM route58_loc12790;

-- STACK 5: min_max_query5
SELECT
    MIN(route_number) AS min_route_num,
    MAX(route_number) AS max_route_num
FROM distinct_routes_at_stop9818;

-- STACK 6: min_max_query6
SELECT
    MIN(stop_time) AS min_stop_time,
    MAX(stop_time) AS max_stop_time
FROM stop_events_with_dow_histogram;

-- STACK 7: min_max_query7
SELECT
    MIN(stop_time) AS min_stop_time,
    MAX(stop_time) AS max_stop_time
FROM model1_v1;

-- STACK 8: min_max_query8
SELECT
    MIN(stop_time) AS min_stop_time,
    MAX(stop_time) AS max_stop_time
FROM model1_v2;

-- STACK 9: min_max_query9
```

```sql
SELECT
    MIN(stop_time) AS min_stop_time,
    MAX(stop_time) AS max_stop_time
FROM model1_v2_compare;

-- STACK 10: min_max_query10
SELECT
    MIN(stop_time) AS min_stop_time,
    MAX(stop_time) AS max_stop_time
FROM model2_v2;

-- STACK 11: min_max_query11
SELECT
    MIN(stop_time) AS min_stop_time,
    MAX(stop_time) AS max_stop_time
FROM model2_v2_2_proj;

-- STACK 12: min_max_query12
SELECT
    MIN(stop_time) AS min_stop_time,
    MAX(stop_time) AS max_stop_time
FROM compare_v2_m1_m2;

-- STACK 13: min_max_query13
SELECT
    MIN(delay_diffs) AS min_delay_diffs,
    MAX(delay_diffs) AS max_delay_diffs
FROM baseline_l8;

-- STACK 14: min_max_query14
SELECT
    MIN(delay) AS min_delay,
```

```sql
    MAX(delay) AS max_delay
FROM baseline_rush_hour_l5;

-- STACK 15: min_max_query15
SELECT
    MIN(delay_diffs) AS min_delay_diffs,
    MAX(delay_diffs) AS max_delay_diffs
FROM predicting_feb_arrival_l11;

-- STACK 16: min_max_query16
SELECT
    MIN(delay_diff) AS min_delay_diff,
    MAX(delay_diff) AS max_delay_diff
FROM predicting_feb_arrival_rush_hr_l8;

-- STACK 17: min_max_query17
SELECT
    MIN(delay_diffs) AS min_delay_diffs,
    MAX(delay_diffs) AS max_delay_diffs
FROM predicting_feb_arrival_dow_class_l9;

-- STACK 18: min_max_query18
SELECT
    MIN(delay_diff) AS min_delay_diff,
    MAX(delay_diff) AS max_delay_diff
FROM predicting_feb_arrival_rush_hr_dow_class_l6;

-- STACK 19: min_max_query19
SELECT
    MIN(stop_time) AS min_stop_time,
    MAX(stop_time) AS max_stop_time
FROM model1_v3_l9;
```

```sql

-- STACK 20: min_max_query20
SELECT
    MIN(stop_time) AS min_stop_time,
    MAX(stop_time) AS max_stop_time
FROM model2_v3_l7;

-- STACK 21: min_max_query21
SELECT
    MIN(prediction_diffs) AS min_prediction_diffs,
    MAX(prediction_diffs) AS max_prediction_diffs
FROM baseline_v2_l9;

-- STACK 22: min_max_query22
SELECT
    MIN(prediction_diffs) AS min_prediction_diffs,
    MAX(prediction_diffs) AS max_prediction_diffs
FROM baseline_v2_rush_hour_l9;

-- STACK 23: min_max_query23
SELECT
    MIN(prediction_diffs) AS min_prediction_diffs,
    MAX(prediction_diffs) AS max_prediction_diffs
FROM comp_predic_v2_l10;

-- STACK 24: min_max_query24
SELECT
    MIN(prediction_diffs) AS min_prediction_diffs,
    MAX(prediction_diffs) AS max_prediction_diffs
FROM comp_predic_v2_rush_hour_l10;

-- STACK 25: min_max_query25
```

```
154 SELECT
155     MIN(prediction_diffs) AS min_prediction_diffs,
156     MAX(prediction_diffs) AS max_prediction_diffs
157 FROM comp_predic_v3_l10;
158
159 -- STACK 26: min_max_query26
160 SELECT
161     MIN(prediction_diffs) AS min_prediction_diffs,
162     MAX(prediction_diffs) AS max_prediction_diffs
163 FROM comp_predic_v3_rush_hour_l10;
164
165 -- STACK 27: min_max_query27
166 SELECT
167     MIN(stop_time) AS min_stop_time,
168     MAX(stop_time) AS max_stop_time
169 FROM comp_pred_model1_and_model2_l4;
```

A.6 POSTGRESQL-EQUIVALENT ANALYSIS

The following is the equivalent analysis using PostgreSQL [28]. Unlike the original analysis, the goal here is to materialize every possible intermediate result to simulate what we did with jSQL$_e$. However, since PostgreSQL does not support in-memory tables (as of Version 12, the latest version during the writing of this dissertation), we just used regular tables (disk-based storage) to see how much effect using disks has on data-access time. We know that using disks is very slow. However, PostgreSQL (as with any typical relational database management system) caches recently used data in memory. So the combination of using disks and caching makes data access much faster. Thus the goal is to test the effect of using disks within that hybrid environment. Note that each of the following statements is a DDL (Data Definition Language) statement, which means the DBMS treats each statement as a separate transaction.

```
CREATE TABLE stop_events
(
    service_date date,
    leave_time integer,
    route_number integer,
    stop_time integer,
    arrive_time integer,
    location_id integer,
    schedule_status integer
);

-- STMT: 1
CREATE TABLE route58_stop910 AS
SELECT * FROM stop_events
WHERE
    service_date = '2018-12-10' AND
    route_number = 58 AND
    location_id = 910;

CREATE TABLE route58_stop910_ordered AS
SELECT * FROM route58_stop910
ORDER BY arrive_time;


-- STMT: 2
CREATE TABLE stop9821 AS
SELECT * FROM stop_events
WHERE
    service_date = '2018-12-10' AND
    location_id = 9821;

CREATE TABLE distinct_routes_at_stop9821 AS
```

```
SELECT DISTINCT route_number FROM stop9821;

-- STMT: 3
CREATE TABLE unique_stops_count AS
SELECT
    service_date, route_number,
    location_id, stop_time,
    count(*) as occurances
FROM stop_events
GROUP BY
    service_date, route_number,
    location_id, stop_time;

CREATE TABLE duplicates AS
SELECT * FROM unique_stops_count
WHERE occurances > 1;


-- STMT: 4
CREATE TABLE route58_loc12790 AS
SELECT * FROM stop_events
WHERE
    service_date = '2018-12-02' AND
    route_number = 58 AND
    location_id = 12790 AND
    stop_time = 38280;


-- STMT: 5
CREATE TABLE stop9818 AS
SELECT * FROM stop_events
WHERE
```

```
    service_date = '2018-12-10' AND
    location_id = 9818;

CREATE TABLE distinct_routes_at_stop9818 AS
SELECT DISTINCT route_number FROM stop9818;


-- STMT: 6
CREATE TABLE stop_events_with_dow AS
SELECT
    t1.*, extract('dow' from service_date) AS day_of_week,
    TRUNC((arrive_time - stop_time) / 60)::int * 60 AS delay,
    CASE
        WHEN extract('dow' from service_date) IN (1,2,3,4,5) THEN 'D'
        WHEN extract('dow' from service_date) = 0 THEN 'U'
        ELSE 'S'
    END AS dow_class
FROM stop_events AS t1;

CREATE TABLE stop_events_with_dow_histogram AS
SELECT
    day_of_week, route_number, location_id,
    stop_time, delay,
    count(*) AS num_of_observations
FROM stop_events_with_dow
GROUP BY
    day_of_week, route_number,
    location_id, stop_time, delay;


-- STMT: 7
CREATE TABLE model1_v1_avg_delay_per_dow AS
```

```
97  SELECT *
98  FROM stop_events_with_dow
99  WHERE
100     service_date >= '2018-11-01' AND
101     service_date < '2018-12-15' OR
102     service_date >= '2019-01-10' AND
103     service_date < '2019-02-01';
104
105 CREATE TABLE model1_v1_agg AS
106 SELECT
107     day_of_week, route_number, location_id, stop_time,
108     AVG(arrive_time - stop_time) AS avg_delay_raw,
109     count(*) AS num_of_observations
110 FROM model1_v1_avg_delay_per_dow
111 GROUP BY
112     day_of_week, route_number,
113     location_id, stop_time;
114
115 CREATE TABLE model1_v1 AS
116 SELECT t1.*, TRUNC(avg_delay_raw)::int AS avg_delay
117 FROM model1_v1_agg AS t1;
118
119
120 -- STMT: 8
121 CREATE TABLE model1_v2_select_base_data AS
122 SELECT *
123 FROM stop_events_with_dow
124 WHERE
125     (
126         service_date >= '2018-12-01' AND
127         service_date < '2018-12-15' OR
128         service_date >= '2019-01-10' AND
```

```
129         service_date < '2019-02-01'
130     ) AND
131     route_number <> 0;
132
133 CREATE TABLE model1_v2_select_base_data_with_delay AS
134 SELECT
135   service_date, leave_time, route_number,
136     stop_time, arrive_time, location_id,
137     schedule_status, day_of_week, dow_class,
138     CASE
139         WHEN ABS(arrive_time - stop_time) <= ABS(leave_time -
                stop_time)
140             THEN arrive_time - stop_time
141         ELSE leave_time - stop_time
142     END AS delay
143 FROM model1_v2_select_base_data AS t1;
144
145 CREATE TABLE model1_v2_cleaned_base_data AS
146 SELECT t1.*
147 FROM
148     model1_v2_select_base_data_with_delay AS t1,
149     (
150         SELECT
151             service_date, day_of_week,
152             route_number, location_id, stop_time,
153             min(delay) AS min_delay
154         FROM model1_v2_select_base_data_with_delay
155         GROUP BY
156             service_date, day_of_week,
157             route_number, location_id, stop_time
158     ) AS t2
159 WHERE
```

```
    t1.delay = t2.min_delay AND
    t1.service_date = t2.service_date AND
    t1.day_of_week = t2.day_of_week AND
    t1.route_number = t2.route_number AND
    t1.location_id = t2.location_id AND
    t1.stop_time = t2.stop_time;

CREATE TABLE model1_v2_base_model AS
SELECT
    day_of_week, route_number,
    location_id, stop_time,
    STDDEV(delay) AS std_delay,
    AVG(delay) AS avg_delay
FROM model1_v2_cleaned_base_data
GROUP BY day_of_week, route_number, location_id, stop_time;

CREATE TABLE model1_v2_final_res_join AS
SELECT
    model1_v2_base_model.day_of_week AS t2_day_of_week,
    model1_v2_base_model.route_number AS t2_route_number,
    model1_v2_base_model.location_id AS t2_location_id,
    model1_v2_base_model.stop_time AS t2_stop_time,
    model1_v2_base_model.std_delay AS t2_std_delay,
    model1_v2_base_model.avg_delay AS t2_avg_delay,
    model1_v2_cleaned_base_data.service_date AS t1_service_date,
    model1_v2_cleaned_base_data.leave_time AS t1_leave_time,
    model1_v2_cleaned_base_data.route_number AS t1_route_number,
    model1_v2_cleaned_base_data.stop_time AS t1_stop_time,
    model1_v2_cleaned_base_data.arrive_time AS t1_arrive_time,
    model1_v2_cleaned_base_data.location_id AS t1_location_id,
    model1_v2_cleaned_base_data.schedule_status AS t1_schedule_status,
    model1_v2_cleaned_base_data.day_of_week AS t1_day_of_week,
```

```
192     model1_v2_cleaned_base_data.dow_class AS t1_dow_class,
193     model1_v2_cleaned_base_data.delay AS t1_delay
194 FROM
195     model1_v2_base_model
196     LEFT JOIN model1_v2_cleaned_base_data ON
197         model1_v2_base_model.day_of_week = model1_v2_cleaned_base_data.
                day_of_week AND
198         model1_v2_base_model.route_number = model1_v2_cleaned_base_data
                .route_number AND
199         model1_v2_base_model.location_id = model1_v2_cleaned_base_data.
                location_id AND
200         model1_v2_base_model.stop_time = model1_v2_cleaned_base_data.
                stop_time AND
201         ABS(model1_v2_cleaned_base_data.delay) <= ABS(
                model1_v2_base_model.avg_delay) + model1_v2_base_model.
                std_delay;
202
203 CREATE TABLE model1_v2_final_res_agg AS
204 SELECT
205     t2_day_of_week, t2_route_number, t2_location_id,
206     t2_stop_time, t2_avg_delay, AVG(t1_delay) AS delay
207 FROM model1_v2_final_res_join
208 GROUP BY
209     t2_day_of_week, t2_route_number,
210     t2_location_id, t2_stop_time, t2_avg_delay;
211
212 CREATE TABLE model1_v2 AS
213 SELECT
214     t2_day_of_week AS day_of_week, t2_route_number AS route_number,
215     t2_location_id AS location_id, t2_stop_time AS stop_time,
216     TRUNC(COALESCE(delay, t2_avg_delay))::int AS avg_delay
217 FROM model1_v2_final_res_agg;
```

```sql

-- STMT: 9
CREATE TABLE model1_v2_compare_sel_route AS
SELECT *
FROM model1_v2
WHERE route_number = 78;

CREATE TABLE model1_v2_compare_sel_dow_tue AS
SELECT *
FROM model1_v2_compare_sel_route
WHERE day_of_week = 2;

CREATE TABLE model1_v2_compare_sel_dow_wed AS
SELECT *
FROM model1_v2_compare_sel_route
WHERE day_of_week = 3;

CREATE TABLE model1_v2_compare_join AS
SELECT
    model1_v2_compare_sel_dow_tue.day_of_week AS t1_day_of_week,
    model1_v2_compare_sel_dow_tue.route_number AS t1_route_number,
    model1_v2_compare_sel_dow_tue.location_id AS t1_location_id,
    model1_v2_compare_sel_dow_tue.stop_time AS t1_stop_time,
    model1_v2_compare_sel_dow_tue.avg_delay AS t1_avg_delay,
    model1_v2_compare_sel_dow_wed.day_of_week AS t2_day_of_week,
    model1_v2_compare_sel_dow_wed.route_number AS t2_route_number,
    model1_v2_compare_sel_dow_wed.location_id AS t2_location_id,
    model1_v2_compare_sel_dow_wed.stop_time AS t2_stop_time,
    model1_v2_compare_sel_dow_wed.avg_delay AS t2_avg_delay
FROM
    model1_v2_compare_sel_dow_tue
```

```
250     JOIN model1_v2_compare_sel_dow_wed ON
251         model1_v2_compare_sel_dow_tue.location_id =
                model1_v2_compare_sel_dow_wed.location_id AND
252         model1_v2_compare_sel_dow_tue.stop_time =
                model1_v2_compare_sel_dow_wed.stop_time;
253
254 CREATE TABLE model1_v2_compare_project AS
255 SELECT
256     t1_route_number AS route_number, t1_location_id AS location_id,
257     t1_stop_time AS stop_time, TRUNC(t1_avg_delay / 60)::int AS
            dow1_delay,
258     TRUNC(t2_avg_delay / 60)::int AS dow2_delay
259 FROM model1_v2_compare_join;
260
261 CREATE TABLE model1_v2_compare AS
262 SELECT *
263 FROM model1_v2_compare_project
264 ORDER BY location_id, stop_time;
265
266
267 -- STMT: 10
268 CREATE TABLE model2_v2_cleaned_base_data AS
269 SELECT t1.*
270 FROM
271     model1_v2_select_base_data_with_delay AS t1,
272     (
273         SELECT
274             service_date, dow_class, route_number,
275             location_id, stop_time, min(delay) AS min_delay
276         FROM model1_v2_select_base_data_with_delay
277         GROUP BY
278             service_date, dow_class,
```

```
            route_number, location_id, stop_time
    ) AS t2
WHERE
    t1.delay = t2.min_delay AND
    t1.service_date = t2.service_date AND
    t1.dow_class = t2.dow_class AND
    t1.route_number = t2.route_number AND
    t1.location_id = t2.location_id AND
    t1.stop_time = t2.stop_time;

CREATE TABLE model2_v2_base_model AS
SELECT
    dow_class, route_number, location_id,
    stop_time, STDDEV(delay) AS std_delay,
    AVG(delay) AS avg_delay
FROM model2_v2_cleaned_base_data
GROUP BY
    dow_class, route_number,
    location_id, stop_time;


CREATE TABLE model2_v2_final_res_join AS
SELECT
    model2_v2_base_model.dow_class AS t2_dow_class,
    model2_v2_base_model.route_number AS t2_route_number,
    model2_v2_base_model.location_id AS t2_location_id,
    model2_v2_base_model.stop_time AS t2_stop_time,
    model2_v2_base_model.std_delay AS t2_std_delay,
    model2_v2_base_model.avg_delay AS t2_avg_delay,
    model2_v2_cleaned_base_data.service_date AS t1_service_date,
    model2_v2_cleaned_base_data.leave_time AS t1_leave_time,
    model2_v2_cleaned_base_data.route_number AS t1_route_number,
```

```
311     model2_v2_cleaned_base_data.stop_time AS t1_stop_time,
312     model2_v2_cleaned_base_data.arrive_time AS t1_arrive_time,
313     model2_v2_cleaned_base_data.location_id AS t1_location_id,
314     model2_v2_cleaned_base_data.schedule_status AS t1_schedule_status,
315     model2_v2_cleaned_base_data.day_of_week AS t1_day_of_week,
316     model2_v2_cleaned_base_data.dow_class AS t1_dow_class,
317     model2_v2_cleaned_base_data.delay AS t1_delay
318 FROM
319     model2_v2_base_model
320     LEFT JOIN model2_v2_cleaned_base_data ON
321         model2_v2_base_model.dow_class = model2_v2_cleaned_base_data.
                dow_class AND
322         model2_v2_base_model.route_number = model2_v2_cleaned_base_data
                .route_number AND
323         model2_v2_base_model.location_id = model2_v2_cleaned_base_data.
                location_id AND
324         model2_v2_base_model.stop_time = model2_v2_cleaned_base_data.
                stop_time AND
325         ABS(model2_v2_cleaned_base_data.delay) <= ABS(
                model2_v2_base_model.avg_delay) + model2_v2_base_model.
                std_delay;
326
327 CREATE TABLE model2_v2_final_res_agg AS
328 SELECT
329     t2_dow_class, t2_route_number,
330     t2_location_id, t2_stop_time,
331     t2_avg_delay, AVG(t1_delay) AS delay
332 FROM model2_v2_final_res_join
333 GROUP BY t2_dow_class, t2_route_number, t2_location_id, t2_stop_time,
        t2_avg_delay;
334
335 CREATE TABLE model2_v2 AS
```

```
SELECT
    t2_dow_class AS dow_class, t2_route_number AS route_number,
    t2_location_id AS location_id, t2_stop_time AS stop_time,
    TRUNC(COALESCE(delay, t2_avg_delay))::int AS avg_delay
FROM model2_v2_final_res_agg;



-- STMT: 11
CREATE TABLE model2_v2_2_avg_delay_per_dow_class AS
SELECT *
FROM stop_events_with_dow
WHERE service_date >= '2018-11-01' AND service_date < '2019-02-01';

CREATE TABLE model2_v2_2_agg AS
SELECT
    dow_class, route_number, location_id, stop_time,
    AVG(arrive_time - stop_time) AS avg_delay_raw, count(*) AS
        num_of_observations
FROM model2_v2_2_avg_delay_per_dow_class
GROUP BY dow_class, route_number, location_id, stop_time;

CREATE TABLE model2_v2_2_proj AS
SELECT t1.*, TRUNC(avg_delay_raw)::int AS avg_delay
FROM model2_v2_2_agg AS t1;


-- STMT: 12
CREATE TABLE compare_v2_m1_m2_sel_m1 AS
SELECT *
FROM model1_v2
WHERE
```

```
367     day_of_week = 5 AND
368     route_number in (76, 78) AND
369     location_id = 2285;
370
371 CREATE TABLE compare_v2_m1_m2_sel_m2 AS
372 SELECT *
373 FROM model2_v2
374 WHERE dow_class = 'D';
375
376 CREATE TABLE compare_v2_m1_m2_join AS
377 SELECT
378     compare_v2_m1_m2_sel_m1.day_of_week AS t1_day_of_week,
379     compare_v2_m1_m2_sel_m1.route_number AS t1_route_number,
380     compare_v2_m1_m2_sel_m1.location_id AS t1_location_id,
381     compare_v2_m1_m2_sel_m1.stop_time AS t1_stop_time,
382     compare_v2_m1_m2_sel_m1.avg_delay AS t1_avg_delay,
383     compare_v2_m1_m2_sel_m2.dow_class AS t2_dow_class,
384     compare_v2_m1_m2_sel_m2.route_number AS t2_route_number,
385     compare_v2_m1_m2_sel_m2.location_id AS t2_location_id,
386     compare_v2_m1_m2_sel_m2.stop_time AS t2_stop_time,
387     compare_v2_m1_m2_sel_m2.avg_delay AS t2_avg_delay
388 FROM
389     compare_v2_m1_m2_sel_m1
390     JOIN compare_v2_m1_m2_sel_m2 ON
391         compare_v2_m1_m2_sel_m1.route_number = compare_v2_m1_m2_sel_m2.
              route_number AND
392         compare_v2_m1_m2_sel_m1.location_id = compare_v2_m1_m2_sel_m2.
              location_id AND
393         compare_v2_m1_m2_sel_m1.stop_time = compare_v2_m1_m2_sel_m2.
              stop_time;
394
395 CREATE TABLE compare_v2_m1_m2_project AS
```

```sql
SELECT
    t1_route_number AS route_number, t1_location_id AS location_id,
    t1_stop_time AS stop_time, TRUNC(t1_avg_delay / 60)::int AS
        dow1_delay,
    TRUNC(t2_avg_delay / 60)::int AS dow2_delay
FROM compare_v2_m1_m2_join;

CREATE TABLE compare_v2_m1_m2 AS
SELECT *
FROM compare_v2_m1_m2_project
ORDER BY location_id, stop_time;


-- STMT: 13
CREATE TABLE baseline_l1 AS
SELECT *
FROM stop_events_with_dow
WHERE service_date >= '2019-02-01' AND service_date < '2019-03-01';

CREATE TABLE baseline_l3 AS
SELECT delay, COUNT(*) AS observations
FROM baseline_l1
GROUP BY delay;

CREATE TABLE baseline_l4 AS
SELECT t1.*, CASE WHEN ABS(delay) > 5 THEN 'others' ELSE delay::text
    END AS delay_diffs
FROM baseline_l3 AS t1;

CREATE TABLE baseline_l6 AS
SELECT delay_diffs, SUM(observations) AS observations
FROM baseline_l4
```

```sql
GROUP BY delay_diffs;

CREATE TABLE baseline_l7 AS
SELECT delay_diffs, observations
FROM baseline_l6;

CREATE TABLE baseline_l8 AS
SELECT *
FROM baseline_l7
ORDER BY delay_diffs;


-- STMT: 14
CREATE TABLE baseline_rush_hour_l1 AS
SELECT *
FROM baseline_l1
WHERE
    stop_time BETWEEN 23160 AND 31140 OR
    stop_time BETWEEN 57600 AND 66780;

CREATE TABLE baseline_rush_hour_l4 AS
SELECT delay, COUNT(*) AS observations
FROM baseline_rush_hour_l1
GROUP BY delay;

CREATE TABLE baseline_rush_hour_l5 AS
SELECT *
FROM baseline_rush_hour_l4
ORDER BY observations DESC;


-- STMT: 15
```

```
458 CREATE TABLE predicting_feb_arrival_l1 AS
459 SELECT *
460 FROM baseline_l1
461 WHERE route_number != 0 AND schedule_status != 6;
462
463 CREATE TABLE predicting_feb_arrival_l2 AS
464 SELECT
465     t1.*,
466     TRUNC(CASE
467         WHEN ABS(arrive_time - stop_time) <= ABS(leave_time -
                stop_time)
468             THEN arrive_time - stop_time
469         ELSE leave_time - stop_time
470     END / 60)::int AS actual_delay_in_min
471 FROM predicting_feb_arrival_l1 AS t1;
472
473 CREATE TABLE predicting_feb_arrival_l3 AS
474 SELECT
475     predicting_feb_arrival_l2.service_date AS t1_service_date,
476     predicting_feb_arrival_l2.leave_time AS t1_leave_time,
477     predicting_feb_arrival_l2.route_number AS t1_route_number,
478     predicting_feb_arrival_l2.stop_time AS t1_stop_time,
479     predicting_feb_arrival_l2.arrive_time AS t1_arrive_time,
480     predicting_feb_arrival_l2.location_id AS t1_location_id,
481     predicting_feb_arrival_l2.schedule_status AS t1_schedule_status,
482     predicting_feb_arrival_l2.day_of_week AS t1_day_of_week,
483     predicting_feb_arrival_l2.delay AS t1_delay,
484     predicting_feb_arrival_l2.dow_class AS t1_dow_class,
485     predicting_feb_arrival_l2.actual_delay_in_min AS
          t1_actual_delay_in_min,
486     model1_v2.day_of_week AS t2_day_of_week,
487     model1_v2.route_number AS t2_route_number,
```

```
488     model1_v2.location_id AS t2_location_id,
489     model1_v2.stop_time AS t2_stop_time,
490     model1_v2.avg_delay AS t2_avg_delay
491 FROM predicting_feb_arrival_l2
492     JOIN model1_v2 ON
493         predicting_feb_arrival_l2.route_number = model1_v2.route_number
                AND
494         predicting_feb_arrival_l2.location_id = model1_v2.location_id
               AND
495         predicting_feb_arrival_l2.stop_time = model1_v2.stop_time AND
496         predicting_feb_arrival_l2.day_of_week = model1_v2.day_of_week;
497
498 CREATE TABLE predicting_feb_arrival_l4 AS
499 SELECT t1.*, TRUNC(t2_avg_delay / 60)::int - t1_actual_delay_in_min AS
        delay_diff
500 FROM predicting_feb_arrival_l3 AS t1;
501
502 CREATE TABLE predicting_feb_arrival_l6 AS
503 SELECT delay_diff, COUNT(*) AS observations
504 FROM predicting_feb_arrival_l4
505 GROUP BY delay_diff;
506
507 CREATE TABLE predicting_feb_arrival_l7 AS
508 SELECT
509     t1.*,
510     CASE
511         WHEN ABS(delay_diff) > 3 THEN 'others'
512         ELSE delay_diff::text
513     END AS delay_diffs
514 FROM predicting_feb_arrival_l6 AS t1;
515
516 CREATE TABLE predicting_feb_arrival_l10 AS
```

```
SELECT delay_diffs, SUM(observations) AS observations
FROM predicting_feb_arrival_l7
GROUP BY delay_diffs;

CREATE TABLE predicting_feb_arrival_l11 AS
SELECT *
FROM predicting_feb_arrival_l10
ORDER BY delay_diffs;


-- STMT: 16
CREATE TABLE predicting_feb_arrival_rush_hr_l1 AS
SELECT *
FROM baseline_l1
WHERE
    stop_time BETWEEN 23160 AND 31140 OR
    stop_time BETWEEN 57600 AND 66780;

CREATE TABLE predicting_feb_arrival_rush_hr_l2 AS
SELECT t1.*, TRUNC(delay / 60)::int AS actual_delay_in_min
FROM predicting_feb_arrival_rush_hr_l1 AS t1;

CREATE TABLE predicting_feb_arrival_rush_hr_l3 AS
SELECT
    predicting_feb_arrival_rush_hr_l2.service_date AS t1_service_date,
    predicting_feb_arrival_rush_hr_l2.leave_time AS t1_leave_time,
    predicting_feb_arrival_rush_hr_l2.route_number AS t1_route_number,
    predicting_feb_arrival_rush_hr_l2.stop_time AS t1_stop_time,
    predicting_feb_arrival_rush_hr_l2.arrive_time AS t1_arrive_time,
    predicting_feb_arrival_rush_hr_l2.location_id AS t1_location_id,
    predicting_feb_arrival_rush_hr_l2.schedule_status AS
        t1_schedule_status,
```

```
548     predicting_feb_arrival_rush_hr_l2.day_of_week AS t1_day_of_week,
549     predicting_feb_arrival_rush_hr_l2.delay AS t1_delay,
550     predicting_feb_arrival_rush_hr_l2.dow_class AS t1_dow_class,
551     predicting_feb_arrival_rush_hr_l2.actual_delay_in_min AS
            t1_actual_delay_in_min,
552     model1_v2.day_of_week AS t2_day_of_week,
553     model1_v2.route_number AS t2_route_number,
554     model1_v2.location_id AS t2_location_id,
555     model1_v2.stop_time AS t2_stop_time,
556     model1_v2.avg_delay AS t2_avg_delay
557 FROM predicting_feb_arrival_rush_hr_l2
558     JOIN model1_v2 ON
559         predicting_feb_arrival_rush_hr_l2.route_number = model1_v2.
                route_number AND
560         predicting_feb_arrival_rush_hr_l2.location_id = model1_v2.
                location_id AND
561         predicting_feb_arrival_rush_hr_l2.stop_time = model1_v2.
                stop_time AND
562         predicting_feb_arrival_rush_hr_l2.day_of_week = model1_v2.
                day_of_week;
563
564 CREATE TABLE predicting_feb_arrival_rush_hr_l4 AS
565 SELECT t1.*, TRUNC(t2_avg_delay / 60)::int - t1_actual_delay_in_min AS
        delay_diff
566 FROM predicting_feb_arrival_rush_hr_l3 AS t1;
567
568 CREATE TABLE predicting_feb_arrival_rush_hr_l7 AS
569 SELECT delay_diff, COUNT(*) AS observations
570 FROM predicting_feb_arrival_rush_hr_l4
571 GROUP BY delay_diff;
572
573 CREATE TABLE predicting_feb_arrival_rush_hr_l8 AS
```

```
SELECT *
FROM predicting_feb_arrival_rush_hr_l7
ORDER BY observations DESC;


-- STMT: 17
CREATE TABLE predicting_feb_arrival_dow_class_l1 AS
SELECT
    predicting_feb_arrival_l2.service_date AS t1_service_date,
    predicting_feb_arrival_l2.leave_time AS t1_leave_time,
    predicting_feb_arrival_l2.route_number AS t1_route_number,
    predicting_feb_arrival_l2.stop_time AS t1_stop_time,
    predicting_feb_arrival_l2.arrive_time AS t1_arrive_time,
    predicting_feb_arrival_l2.location_id AS t1_location_id,
    predicting_feb_arrival_l2.schedule_status AS t1_schedule_status,
    predicting_feb_arrival_l2.day_of_week AS t1_day_of_week,
    predicting_feb_arrival_l2.delay AS t1_delay,
    predicting_feb_arrival_l2.dow_class AS t1_dow_class,
    predicting_feb_arrival_l2.actual_delay_in_min AS
        t1_actual_delay_in_min,
    model2_v2_2_proj.dow_class AS t2_dow_class,
    model2_v2_2_proj.route_number AS t2_route_number,
    model2_v2_2_proj.location_id AS t2_location_id,
    model2_v2_2_proj.stop_time AS t2_stop_time,
    model2_v2_2_proj.avg_delay_raw AS t2_avg_delay_raw,
    model2_v2_2_proj.num_of_observations AS t2_num_of_observations,
    model2_v2_2_proj.avg_delay AS t2_avg_delay
FROM predicting_feb_arrival_l2
    JOIN model2_v2_2_proj ON
        predicting_feb_arrival_l2.route_number = model2_v2_2_proj.
            route_number AND
```

```
603         predicting_feb_arrival_l2.location_id = model2_v2_2_proj.
                location_id AND
604         predicting_feb_arrival_l2.stop_time = model2_v2_2_proj.
                stop_time AND
605         predicting_feb_arrival_l2.dow_class = model2_v2_2_proj.
                dow_class;
606
607 CREATE TABLE predicting_feb_arrival_dow_class_l2 AS
608 SELECT
609     t1.*,
610     TRUNC(t2_avg_delay / 60)::int - t1_actual_delay_in_min AS
            delay_diff
611 FROM predicting_feb_arrival_dow_class_l1 AS t1;
612
613 CREATE TABLE predicting_feb_arrival_dow_class_l4 AS
614 SELECT delay_diff, COUNT(*) AS observations
615 FROM predicting_feb_arrival_dow_class_l2
616 GROUP BY delay_diff;
617
618 CREATE TABLE predicting_feb_arrival_dow_class_l5 AS
619 SELECT
620     t1.*,
621     CASE
622         WHEN ABS(delay_diff) > 3 THEN 'others'
623         ELSE delay_diff::text
624     END AS delay_diffs
625 FROM predicting_feb_arrival_dow_class_l4 AS t1;
626
627 CREATE TABLE predicting_feb_arrival_dow_class_l8 AS
628 SELECT delay_diffs, SUM(observations) AS observations
629 FROM predicting_feb_arrival_dow_class_l5
630 GROUP BY delay_diffs;
```

```
631
632 CREATE TABLE predicting_feb_arrival_dow_class_l9 AS
633 SELECT *
634 FROM predicting_feb_arrival_dow_class_l8
635 ORDER BY delay_diffs;
636
637
638 -- STMT: 18
639 CREATE TABLE predicting_feb_arrival_rush_hr_dow_class_l1 AS
640 SELECT
641     predicting_feb_arrival_rush_hr_l2.service_date AS t1_service_date,
642     predicting_feb_arrival_rush_hr_l2.leave_time AS t1_leave_time,
643     predicting_feb_arrival_rush_hr_l2.route_number AS t1_route_number,
644     predicting_feb_arrival_rush_hr_l2.stop_time AS t1_stop_time,
645     predicting_feb_arrival_rush_hr_l2.arrive_time AS t1_arrive_time,
646     predicting_feb_arrival_rush_hr_l2.location_id AS t1_location_id,
647     predicting_feb_arrival_rush_hr_l2.schedule_status AS
            t1_schedule_status,
648     predicting_feb_arrival_rush_hr_l2.day_of_week AS t1_day_of_week,
649     predicting_feb_arrival_rush_hr_l2.delay AS t1_delay,
650     predicting_feb_arrival_rush_hr_l2.dow_class AS t1_dow_class,
651     predicting_feb_arrival_rush_hr_l2.actual_delay_in_min AS
            t1_actual_delay_in_min,
652     model2_v2_2_proj.dow_class AS t2_dow_class,
653     model2_v2_2_proj.route_number AS t2_route_number,
654     model2_v2_2_proj.location_id AS t2_location_id,
655     model2_v2_2_proj.stop_time AS t2_stop_time,
656     model2_v2_2_proj.avg_delay_raw AS t2_avg_delay_raw,
657     model2_v2_2_proj.num_of_observations AS t2_num_of_observations,
658     model2_v2_2_proj.avg_delay AS t2_avg_delay
659 FROM predicting_feb_arrival_rush_hr_l2
660     JOIN model2_v2_2_proj ON
```

```
661         predicting_feb_arrival_rush_hr_l2.route_number =
                model2_v2_2_proj.route_number AND
662         predicting_feb_arrival_rush_hr_l2.location_id =
                model2_v2_2_proj.location_id AND
663         predicting_feb_arrival_rush_hr_l2.stop_time = model2_v2_2_proj.
                stop_time AND
664         predicting_feb_arrival_rush_hr_l2.dow_class = model2_v2_2_proj.
                dow_class;
665
666 CREATE TABLE predicting_feb_arrival_rush_hr_dow_class_l2 AS
667 SELECT t1.*, TRUNC(t2_avg_delay / 60)::int - t1_actual_delay_in_min AS
        delay_diff
668 FROM predicting_feb_arrival_rush_hr_dow_class_l1 AS t1;
669
670 CREATE TABLE predicting_feb_arrival_rush_hr_dow_class_l5 AS
671 SELECT delay_diff, COUNT(*) AS observations
672 FROM predicting_feb_arrival_rush_hr_dow_class_l2
673 GROUP BY delay_diff;
674
675 CREATE TABLE predicting_feb_arrival_rush_hr_dow_class_l6 AS
676 SELECT *
677 FROM predicting_feb_arrival_rush_hr_dow_class_l5
678 ORDER BY observations DESC;
679
680
681 -- STMT: 19
682 CREATE TABLE model1_v3_l2 AS
683 SELECT
684     service_date, route_number, location_id, stop_time,
685     MAX(arrive_time) AS arrive_time, MAX(leave_time) AS leave_time
686 FROM model1_v2_select_base_data
687 GROUP BY service_date, route_number, location_id, stop_time;
```

```
688
689 CREATE TABLE model1_v3_l3 AS
690 SELECT t1.*, extract('dow' from service_date) AS day_of_week
691 FROM model1_v3_l2 AS t1;
692
693 CREATE TABLE model1_v3_l5 AS
694 SELECT
695     day_of_week, route_number, location_id, stop_time,
696     STDDEV(arrive_time) AS std_arrive_time,
697     AVG(arrive_time) AS avg_arrive_time,
698     STDDEV(leave_time) AS std_leave_time,
699     AVG(leave_time) AS avg_leave_time
700 FROM model1_v3_l3
701 GROUP BY
702     day_of_week, route_number,
703     location_id, stop_time;
704
705 CREATE TABLE model1_v3_l6 AS
706 SELECT
707     model1_v3_l3.service_date AS t1_service_date,
708     model1_v3_l3.route_number AS t1_route_number,
709     model1_v3_l3.location_id AS t1_location_id,
710     model1_v3_l3.stop_time AS t1_stop_time,
711     model1_v3_l3.arrive_time AS t1_arrive_time,
712     model1_v3_l3.leave_time AS t1_leave_time,
713     model1_v3_l3.day_of_week AS t1_day_of_week,
714     model1_v3_l5.day_of_week AS t2_day_of_week,
715     model1_v3_l5.route_number AS t2_route_number,
716     model1_v3_l5.location_id AS t2_location_id,
717     model1_v3_l5.stop_time AS t2_stop_time,
718     model1_v3_l5.std_arrive_time AS t2_std_arrive_time,
719     model1_v3_l5.avg_arrive_time AS t2_avg_arrive_time,
```

```
    model1_v3_l5.std_leave_time AS t2_std_leave_time,
    model1_v3_l5.avg_leave_time AS t2_avg_leave_time
FROM model1_v3_l3
    JOIN model1_v3_l5 ON
        model1_v3_l5.day_of_week = model1_v3_l3.day_of_week AND
        model1_v3_l5.route_number = model1_v3_l3.route_number AND
        model1_v3_l5.location_id = model1_v3_l3.location_id AND
        model1_v3_l5.stop_time = model1_v3_l3.stop_time;

CREATE TABLE model1_v3_l8 AS
SELECT
    t2_day_of_week, t2_route_number, t2_location_id,
    t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time,
    AVG(t1_arrive_time) FILTER(
        WHERE abs(t1_arrive_time) <= abs(t2_avg_arrive_time) +
            t2_std_arrive_time
    ) AS avg_arrive_time,
    AVG(t1_leave_time) FILTER(
        WHERE abs(t1_leave_time) <= abs(t2_avg_leave_time) +
            t2_std_leave_time
    ) AS avg_leave_time
FROM model1_v3_l6
GROUP BY
    t2_day_of_week, t2_route_number, t2_location_id,
    t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time;

CREATE TABLE model1_v3_l9 AS
SELECT
    t2_day_of_week AS day_of_week, t2_route_number AS route_number,
    t2_location_id AS location_id, t2_stop_time AS stop_time,
    TRUNC(COALESCE(avg_arrive_time, t2_avg_arrive_time))::int AS
        arrive_time,
```

```
749     TRUNC(COALESCE(avg_leave_time,t2_avg_leave_time))::int AS
            leave_time
750 FROM model1_v3_l8;
751
752
753 -- STMT: 20
754 CREATE TABLE model2_v3_l1 AS
755 SELECT t1.*,
756     CASE
757         WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
758         WHEN day_of_week = 0 THEN 'U'
759         ELSE 'S'
760     END AS dow_class
761 FROM model1_v3_l3 AS t1;
762
763 CREATE TABLE model2_v3_l3 AS
764 SELECT
765     dow_class, route_number, location_id, stop_time,
766     STDDEV(arrive_time) AS std_arrive_time,
767     AVG(arrive_time) AS avg_arrive_time,
768     STDDEV(leave_time) AS std_leave_time,
769     AVG(leave_time) AS avg_leave_time
770 FROM model2_v3_l1
771 GROUP BY
772     dow_class, route_number,
773     location_id, stop_time;
774
775 CREATE TABLE model2_v3_l4 AS
776 SELECT
777     model2_v3_l1.service_date AS t1_service_date,
778     model2_v3_l1.route_number AS t1_route_number,
779     model2_v3_l1.location_id AS t1_location_id,
```

```
    model2_v3_l1.stop_time AS t1_stop_time,
    model2_v3_l1.arrive_time AS t1_arrive_time,
    model2_v3_l1.leave_time AS t1_leave_time,
    model2_v3_l1.day_of_week AS t1_day_of_week,
    model2_v3_l1.dow_class AS t1_dow_class,
    model2_v3_l3.dow_class AS t2_dow_class,
    model2_v3_l3.route_number AS t2_route_number,
    model2_v3_l3.location_id AS t2_location_id,
    model2_v3_l3.stop_time AS t2_stop_time,
    model2_v3_l3.std_arrive_time AS t2_std_arrive_time,
    model2_v3_l3.avg_arrive_time AS t2_avg_arrive_time,
    model2_v3_l3.std_leave_time AS t2_std_leave_time,
    model2_v3_l3.avg_leave_time AS t2_avg_leave_time
FROM model2_v3_l1
    JOIN model2_v3_l3 ON
        model2_v3_l1.dow_class = model2_v3_l3.dow_class AND
        model2_v3_l1.route_number = model2_v3_l3.route_number AND
        model2_v3_l1.location_id = model2_v3_l3.location_id AND
        model2_v3_l1.stop_time = model2_v3_l3.stop_time;

CREATE TABLE model2_v3_l6 AS
SELECT
    t2_dow_class, t2_route_number, t2_location_id,
    t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time,
    AVG(t1_arrive_time) FILTER(
        WHERE abs(t1_arrive_time) <= abs(t2_avg_arrive_time) +
            t2_std_arrive_time
    ) AS avg_arrive_time,
    AVG(t1_leave_time) FILTER(
        WHERE abs(t1_leave_time) <= abs(t2_avg_leave_time) +
            t2_std_leave_time
    ) AS avg_leave_time
```

```
FROM model2_v3_l4
GROUP BY
    t2_dow_class, t2_route_number, t2_location_id,
    t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time;

CREATE TABLE model2_v3_l7 AS
SELECT
    t2_dow_class AS dow_class, t2_route_number AS route_number,
    t2_location_id AS location_id, t2_stop_time AS stop_time,
    TRUNC(COALESCE(avg_arrive_time, t2_avg_arrive_time))::int AS
        arrive_time,
    TRUNC(COALESCE(avg_leave_time,t2_avg_leave_time))::int AS
        leave_time
FROM model2_v3_l6;


-- STMT: 21
CREATE TABLE baseline_v2_l1 AS
SELECT *
FROM baseline_l1
WHERE route_number <> 0;

CREATE TABLE baseline_v2_l3 AS
SELECT
    service_date, route_number, location_id, stop_time,
    MIN(arrive_time) AS arrive_time, MAX(leave_time) AS leave_time
FROM baseline_v2_l1
GROUP BY service_date, route_number, location_id, stop_time;

CREATE TABLE baseline_v2_l4 AS
SELECT
    TRUNC(CASE
```

```
        WHEN ABS(arrive_time - stop_time) <= ABS(leave_time - 30 -
            stop_time)
            THEN arrive_time - stop_time
        ELSE leave_time - 30 - stop_time
    END / 60)::int AS prediction_diff
FROM baseline_v2_l3;

CREATE TABLE baseline_v2_l5 AS
SELECT
    CASE
        WHEN prediction_diff > 3 THEN 'others'
        ELSE prediction_diff::text
    END AS prediction_diffs
FROM baseline_v2_l4;

CREATE TABLE baseline_v2_l8 AS
SELECT prediction_diffs, COUNT(*) AS observations
FROM baseline_v2_l5
GROUP BY prediction_diffs;

CREATE TABLE baseline_v2_l9 AS
SELECT *
FROM baseline_v2_l8
ORDER BY prediction_diffs;


-- STMT: 22
CREATE TABLE baseline_v2_rush_hour_l1 AS
SELECT *
FROM baseline_rush_hour_l1
WHERE route_number <> 0;

```

```
CREATE TABLE baseline_v2_rush_hour_l3 AS
SELECT
    service_date, route_number, location_id, stop_time,
    MIN(arrive_time) AS arrive_time, MAX(leave_time) AS leave_time
FROM baseline_v2_rush_hour_l1
GROUP BY service_date, route_number, location_id, stop_time;

CREATE TABLE baseline_v2_rush_hour_l4 AS
SELECT
    TRUNC(CASE
        WHEN ABS(arrive_time - stop_time) <= ABS(leave_time - 30 -
            stop_time)
            THEN arrive_time - stop_time
        ELSE leave_time - 30 - stop_time
    END / 60)::int AS prediction_diff
FROM baseline_v2_rush_hour_l3;

CREATE TABLE baseline_v2_rush_hour_l5 AS
SELECT
    CASE
        WHEN prediction_diff > 3 THEN 'others'
        ELSE prediction_diff::text
    END AS prediction_diffs
FROM baseline_v2_rush_hour_l4;

CREATE TABLE baseline_v2_rush_hour_l8 AS
SELECT prediction_diffs, COUNT(*) AS observations
FROM baseline_v2_rush_hour_l5
GROUP BY prediction_diffs;

CREATE TABLE baseline_v2_rush_hour_l9 AS
SELECT *
```

```
FROM baseline_v2_rush_hour_l8
ORDER BY prediction_diffs;


-- STMT: 23
CREATE TABLE comp_predic_v2_l1 AS
SELECT t1.*, extract('dow' from service_date) AS day_of_week
FROM baseline_v2_l3 AS t1;

CREATE TABLE comp_predic_v2_l2 AS
SELECT
    comp_predic_v2_l1.service_date AS t1_service_date,
    comp_predic_v2_l1.route_number AS t1_route_number,
    comp_predic_v2_l1.location_id AS t1_location_id,
    comp_predic_v2_l1.stop_time AS t1_stop_time,
    comp_predic_v2_l1.arrive_time AS t1_arrive_time,
    comp_predic_v2_l1.leave_time AS t1_leave_time,
    comp_predic_v2_l1.day_of_week AS t1_day_of_week,
    model1_v3_l9.day_of_week AS t2_day_of_week,
    model1_v3_l9.route_number AS t2_route_number,
    model1_v3_l9.location_id AS t2_location_id,
    model1_v3_l9.stop_time AS t2_stop_time,
    model1_v3_l9.arrive_time AS t2_arrive_time,
    model1_v3_l9.leave_time AS t2_leave_time
FROM comp_predic_v2_l1
    JOIN model1_v3_l9 ON
        comp_predic_v2_l1.route_number = model1_v3_l9.route_number AND
        comp_predic_v2_l1.location_id = model1_v3_l9.location_id AND
        comp_predic_v2_l1.stop_time = model1_v3_l9.stop_time AND
        comp_predic_v2_l1.day_of_week = model1_v3_l9.day_of_week;

CREATE TABLE comp_predic_v2_l3 AS
```

```
SELECT t1.*, TRUNC((t2_arrive_time - t1_arrive_time) / 60)::int AS
    prediction_diff
FROM comp_predic_v2_l2 AS t1;

CREATE TABLE comp_predic_v2_l5 AS
SELECT prediction_diff, COUNT(*) AS observations
FROM comp_predic_v2_l3
GROUP BY prediction_diff;

CREATE TABLE comp_predic_v2_l6 AS
SELECT t1.*, CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE prediction_diff::text
    END AS prediction_diffs
FROM comp_predic_v2_l5 AS t1;

CREATE TABLE comp_predic_v2_l9 AS
SELECT prediction_diffs, SUM(observations) AS observations
FROM comp_predic_v2_l6
GROUP BY prediction_diffs;

CREATE TABLE comp_predic_v2_l10 AS
SELECT *
FROM comp_predic_v2_l9
ORDER BY prediction_diffs;


-- STMT: 24
CREATE TABLE comp_predic_v2_rush_hour_l1 AS
SELECT t1.*, extract('dow' from service_date) AS day_of_week
FROM baseline_v2_rush_hour_l3 AS t1;

```

```sql
CREATE TABLE comp_predic_v2_rush_hour_l2 AS
SELECT
    comp_predic_v2_rush_hour_l1.service_date AS t1_service_date,
    comp_predic_v2_rush_hour_l1.route_number AS t1_route_number,
    comp_predic_v2_rush_hour_l1.location_id AS t1_location_id,
    comp_predic_v2_rush_hour_l1.stop_time AS t1_stop_time,
    comp_predic_v2_rush_hour_l1.arrive_time AS t1_arrive_time,
    comp_predic_v2_rush_hour_l1.leave_time AS t1_leave_time,
    comp_predic_v2_rush_hour_l1.day_of_week AS t1_day_of_week,
    model1_v3_l9.day_of_week AS t2_day_of_week,
    model1_v3_l9.route_number AS t2_route_number,
    model1_v3_l9.location_id AS t2_location_id,
    model1_v3_l9.stop_time AS t2_stop_time,
    model1_v3_l9.arrive_time AS t2_arrive_time,
    model1_v3_l9.leave_time AS t2_leave_time
FROM comp_predic_v2_rush_hour_l1
    JOIN model1_v3_l9 ON
        comp_predic_v2_rush_hour_l1.route_number = model1_v3_l9.
            route_number AND
        comp_predic_v2_rush_hour_l1.location_id = model1_v3_l9.
            location_id AND
        comp_predic_v2_rush_hour_l1.stop_time = model1_v3_l9.stop_time
            AND
        comp_predic_v2_rush_hour_l1.day_of_week = model1_v3_l9.
            day_of_week;

CREATE TABLE comp_predic_v2_rush_hour_l3 AS
SELECT t1.*, TRUNC((t2_arrive_time - t1_arrive_time) / 60)::int AS
    prediction_diff
FROM comp_predic_v2_rush_hour_l2 AS t1;

CREATE TABLE comp_predic_v2_rush_hour_l5 AS
```

```sql
SELECT prediction_diff, COUNT(*) AS observations
FROM comp_predic_v2_rush_hour_l3
GROUP BY prediction_diff;

CREATE TABLE comp_predic_v2_rush_hour_l6 AS
SELECT t1.*, CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE prediction_diff::text
    END AS prediction_diffs
FROM comp_predic_v2_rush_hour_l5 AS t1;

CREATE TABLE comp_predic_v2_rush_hour_l9 AS
SELECT prediction_diffs, SUM(observations) AS observations
FROM comp_predic_v2_rush_hour_l6
GROUP BY prediction_diffs;

CREATE TABLE comp_predic_v2_rush_hour_l10 AS
SELECT *
FROM comp_predic_v2_rush_hour_l9
ORDER BY prediction_diffs;


-- STMT: 25
CREATE TABLE comp_predic_v3_l1 AS
SELECT t1.*, CASE
    WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
    WHEN day_of_week = 0 THEN 'U'
    ELSE 'S'
    END AS dow_class
FROM comp_predic_v2_l1 AS t1;

CREATE TABLE comp_predic_v3_l2 AS
```

```
SELECT
    comp_predic_v3_l1.service_date AS t1_service_date,
    comp_predic_v3_l1.route_number AS t1_route_number,
    comp_predic_v3_l1.location_id AS t1_location_id,
    comp_predic_v3_l1.stop_time AS t1_stop_time,
    comp_predic_v3_l1.arrive_time AS t1_arrive_time,
    comp_predic_v3_l1.leave_time AS t1_leave_time,
    comp_predic_v3_l1.day_of_week AS t1_day_of_week,
    comp_predic_v3_l1.dow_class AS t1_dow_class,
    model2_v3_l7.dow_class AS t2_dow_class,
    model2_v3_l7.route_number AS t2_route_number,
    model2_v3_l7.location_id AS t2_location_id,
    model2_v3_l7.stop_time AS t2_stop_time,
    model2_v3_l7.arrive_time AS t2_arrive_time,
    model2_v3_l7.leave_time AS t2_leave_time
FROM comp_predic_v3_l1
    JOIN model2_v3_l7 ON
        comp_predic_v3_l1.route_number = model2_v3_l7.route_number AND
        comp_predic_v3_l1.location_id = model2_v3_l7.location_id AND
        comp_predic_v3_l1.stop_time = model2_v3_l7.stop_time AND
        comp_predic_v3_l1.dow_class = model2_v3_l7.dow_class;

CREATE TABLE comp_predic_v3_l3 AS
SELECT
    t1.*,
    TRUNC((t2_arrive_time - t1_arrive_time) / 60)::int AS
        prediction_diff
FROM comp_predic_v3_l2 AS t1;

CREATE TABLE comp_predic_v3_l5 AS
SELECT prediction_diff, COUNT(*) AS observations
FROM comp_predic_v3_l3
```

```
GROUP BY prediction_diff;

CREATE TABLE comp_predic_v3_l6 AS
SELECT t1.*, CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE prediction_diff::text
    END AS prediction_diffs
FROM comp_predic_v3_l5 AS t1;

CREATE TABLE comp_predic_v3_l9 AS
SELECT prediction_diffs, SUM(observations) AS observations
FROM comp_predic_v3_l6
GROUP BY prediction_diffs;

CREATE TABLE comp_predic_v3_l10 AS
SELECT *
FROM comp_predic_v3_l9
ORDER BY prediction_diffs;


-- STMT: 26
CREATE TABLE comp_predic_v3_rush_hour_l1 AS
SELECT t1.*, CASE
    WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
    WHEN day_of_week = 0 THEN 'U'
    ELSE 'S'
    END AS dow_class
FROM comp_predic_v2_rush_hour_l1 AS t1;

CREATE TABLE comp_predic_v3_rush_hour_l2 AS
SELECT
    comp_predic_v3_rush_hour_l1.service_date AS t1_service_date,
```

```
    comp_predic_v3_rush_hour_l1.route_number AS t1_route_number,
    comp_predic_v3_rush_hour_l1.location_id AS t1_location_id,
    comp_predic_v3_rush_hour_l1.stop_time AS t1_stop_time,
    comp_predic_v3_rush_hour_l1.arrive_time AS t1_arrive_time,
    comp_predic_v3_rush_hour_l1.leave_time AS t1_leave_time,
    comp_predic_v3_rush_hour_l1.day_of_week AS t1_day_of_week,
    comp_predic_v3_rush_hour_l1.dow_class AS t1_dow_class,
    model2_v3_l7.dow_class AS t2_dow_class,
    model2_v3_l7.route_number AS t2_route_number,
    model2_v3_l7.location_id AS t2_location_id,
    model2_v3_l7.stop_time AS t2_stop_time,
    model2_v3_l7.arrive_time AS t2_arrive_time,
    model2_v3_l7.leave_time AS t2_leave_time
FROM comp_predic_v3_rush_hour_l1
    JOIN model2_v3_l7 ON
        comp_predic_v3_rush_hour_l1.route_number = model2_v3_l7.
            route_number AND
        comp_predic_v3_rush_hour_l1.location_id = model2_v3_l7.
            location_id AND
        comp_predic_v3_rush_hour_l1.stop_time = model2_v3_l7.stop_time
            AND
        comp_predic_v3_rush_hour_l1.dow_class = model2_v3_l7.dow_class;

CREATE TABLE comp_predic_v3_rush_hour_l3 AS
SELECT
    t1.*,
    TRUNC((t2_arrive_time - t1_arrive_time) / 60)::int AS
        prediction_diff
FROM comp_predic_v3_rush_hour_l2 AS t1;

CREATE TABLE comp_predic_v3_rush_hour_l5 AS
SELECT prediction_diff, COUNT(*) AS observations
```

```sql
FROM comp_predic_v3_rush_hour_l3
GROUP BY prediction_diff;

CREATE TABLE comp_predic_v3_rush_hour_l6 AS
SELECT t1.*, CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE prediction_diff::text
    END AS prediction_diffs
FROM comp_predic_v3_rush_hour_l5 AS t1;

CREATE TABLE comp_predic_v3_rush_hour_l9 AS
SELECT prediction_diffs, SUM(observations) AS observations
FROM comp_predic_v3_rush_hour_l6
GROUP BY prediction_diffs;

CREATE TABLE comp_predic_v3_rush_hour_l10 AS
SELECT *
FROM comp_predic_v3_rush_hour_l9
ORDER BY prediction_diffs;


-- STMT: 27
CREATE TABLE comp_pred_model1_and_model2_l1 AS
SELECT
    model1_v3_l9.day_of_week AS t1_day_of_week,
    model1_v3_l9.route_number AS t1_route_number,
    model1_v3_l9.location_id AS t1_location_id,
    model1_v3_l9.stop_time AS t1_stop_time,
    model1_v3_l9.arrive_time AS t1_arrive_time,
    model1_v3_l9.leave_time AS t1_leave_time,
    model2_v3_l7.dow_class AS t2_dow_class,
    model2_v3_l7.route_number AS t2_route_number,
```

```
    model2_v3_l7.location_id AS t2_location_id,
    model2_v3_l7.stop_time AS t2_stop_time,
    model2_v3_l7.arrive_time AS t2_arrive_time,
    model2_v3_l7.leave_time AS t2_leave_time
FROM model1_v3_l9
    JOIN model2_v3_l7 ON
        model1_v3_l9.route_number = model2_v3_l7.route_number AND
        model1_v3_l9.location_id = model2_v3_l7.location_id AND
        model1_v3_l9.stop_time = model2_v3_l7.stop_time;

CREATE TABLE comp_pred_model1_and_model2_l2 AS
SELECT *
FROM comp_pred_model1_and_model2_l1
WHERE
    t1_day_of_week = 5 AND
    t2_dow_class = 'D' AND
    t1_route_number in (76, 78) AND
    t1_location_id = 2285;

CREATE TABLE comp_pred_model1_and_model2_l3 AS
SELECT
    t1_route_number AS route_number,
    t1_location_id AS location_id,
    t1_stop_time AS stop_time,
    t1_arrive_time AS model1_pred_arrival_time,
    t1_leave_time AS model1_pred_leave_time,
    t2_arrive_time AS model2_pred_arrival_time,
    t2_leave_time AS model2_pred_leave_time
FROM comp_pred_model1_and_model2_l2;

CREATE TABLE comp_pred_model1_and_model2_l4 AS
SELECT *
```

```
1179 FROM comp_pred_model1_and_model2_l3
1180 ORDER BY location_id, stop_time;
```

A.6.1 Min-Max Queries

The min-max queries that we used in PostgreSQL are exactly the same as those we did for MySQL in Section A.5.1.

A.7 SPARK-EQUIVALENT ANALYSIS

The following is the equivalent analysis using Spark [71]. The goal here is to cache every possible intermediate result to simulate what we did with jSQL$_e$. We used Spark's `DataFrames` so that we can model the data as tables and we can use standard SQL to write the queries. We also forced Spark to persist in memory every intermediate result.

```
1  -- STMT: 1
2  -- VIEW NAME: route58_stop910
3  SELECT * FROM stop_events
4  WHERE
5      service_date = '2018-12-10' AND
6      route_number = 58 AND
7      location_id = 910;
8
9  -- VIEW NAME: route58_stop910_ordered
10 SELECT * FROM route58_stop910
11 ORDER BY arrive_time;
12
13
14 -- STMT: 2
15 -- VIEW NAME: stop9821
16 SELECT * FROM stop_events
17 WHERE
```

```
18      service_date = '2018-12-10' AND
19      location_id = 9821;
20
21  -- VIEW NAME: distinct_routes_at_stop9821
22  SELECT DISTINCT route_number FROM stop9821;
23
24
25  -- STMT: 3
26  -- VIEW NAME: unique_stops_count
27  SELECT
28      service_date, route_number, location_id,
29      stop_time, count(*) AS occurances
30  FROM stop_events
31  GROUP BY
32      service_date, route_number,
33      location_id, stop_time;
34
35  -- VIEW NAME: duplicates
36  SELECT * FROM unique_stops_count
37  WHERE occurances > 1;
38
39
40  -- STMT: 4
41  -- VIEW NAME: route58_loc12790
42  SELECT * FROM stop_events
43  WHERE
44      service_date = '2018-12-02' AND
45      route_number = 58 AND
46      location_id = 12790 AND
47      stop_time = 38280;
48
49
```

```sql
-- STMT: 5
-- VIEW NAME: stop9818
SELECT * FROM stop_events
WHERE
    service_date = '2018-12-10' AND
    location_id = 9818;

-- VIEW NAME: distinct_routes_at_stop9818
SELECT DISTINCT route_number FROM stop9818;


-- STMT: 6
-- VIEW NAME: stop_events_with_dow
SELECT
    t1.*, DAYOFWEEK(service_date) - 1 AS day_of_week,
    CAST((arrive_time - stop_time) / 60 AS INT) * 60 AS delay,
    CASE
        WHEN DAYOFWEEK(service_date) - 1 IN (1, 2,3,4,5) THEN 'D'
        WHEN DAYOFWEEK(service_date) - 1 = 1 THEN 'U'
        ELSE 'S'
    END AS dow_class
FROM stop_events AS t1;

-- VIEW NAME: stop_events_with_dow_histogram
SELECT
    day_of_week, route_number, location_id,
    stop_time, delay,
    count(*) AS num_of_observations
FROM stop_events_with_dow
GROUP BY
    day_of_week, route_number,
    location_id, stop_time, delay
```

```sql

-- STMT: 7
-- VIEW NAME: model1_v1_avg_delay_per_dow
SELECT *
FROM stop_events_with_dow
WHERE
    service_date >= '2018-11-01' AND
    service_date < '2018-12-15' OR
    service_date >= '2019-01-10' AND
    service_date < '2019-02-01';

-- VIEW NAME: model1_v1_agg
SELECT
    day_of_week, route_number,
    location_id, stop_time,
    AVG(arrive_time - stop_time) AS avg_delay_raw,
    count(*) AS num_of_observations
FROM model1_v1_avg_delay_per_dow
GROUP BY
    day_of_week, route_number,
    location_id, stop_time;

-- VIEW NAME: model1_v1
SELECT t1.*, CAST(avg_delay_raw AS INT) AS avg_delay
FROM model1_v1_agg AS t1;


-- STMT: 8
-- VIEW NAME: model1_v2_select_base_data
SELECT *
FROM stop_events_with_dow
```

```
114 WHERE
115     (
116         service_date >= '2018-12-01' AND
117         service_date < '2018-12-15' OR
118         service_date >= '2019-01-10' AND
119         service_date < '2019-02-01'
120     ) AND
121     route_number <> 0;
122
123 -- VIEW NAME: model1_v2_select_base_data_with_delay
124 SELECT
125   service_date, leave_time, route_number,
126     stop_time, arrive_time, location_id,
127     schedule_status, day_of_week, dow_class,
128     CASE
129         WHEN ABS(arrive_time - stop_time) <= ABS(leave_time -
                stop_time)
130             THEN arrive_time - stop_time
131         ELSE leave_time - stop_time
132     END AS delay
133 FROM model1_v2_select_base_data AS t1;
134
135 -- VIEW NAME: model1_v2_cleaned_base_data
136 SELECT t1.*
137 FROM
138     model1_v2_select_base_data_with_delay AS t1,
139     (
140         SELECT
141             service_date, day_of_week,
142             route_number, location_id, stop_time,
143             min(delay) AS min_delay
144         FROM model1_v2_select_base_data_with_delay
```

```
145         GROUP BY
146             service_date, day_of_week,
147             route_number, location_id, stop_time
148     ) AS t2
149 WHERE
150     t1.delay = t2.min_delay AND
151     t1.service_date = t2.service_date AND
152     t1.day_of_week = t2.day_of_week AND
153     t1.route_number = t2.route_number AND
154     t1.location_id = t2.location_id AND
155     t1.stop_time = t2.stop_time;
156
157 -- VIEW NAME: model1_v2_base_model
158 SELECT
159     day_of_week, route_number,
160     location_id, stop_time,
161     STDDEV(delay) AS std_delay,
162     AVG(delay) AS avg_delay
163 FROM model1_v2_cleaned_base_data
164 GROUP BY
165     day_of_week, route_number,
166     location_id, stop_time;
167
168 -- VIEW NAME: model1_v2_final_res_join
169 SELECT
170     model1_v2_base_model.day_of_week AS t2_day_of_week,
171     model1_v2_base_model.route_number AS t2_route_number,
172     model1_v2_base_model.location_id AS t2_location_id,
173     model1_v2_base_model.stop_time AS t2_stop_time,
174     model1_v2_base_model.std_delay AS t2_std_delay,
175     model1_v2_base_model.avg_delay AS t2_avg_delay,
176     model1_v2_cleaned_base_data.service_date AS t1_service_date,
```

```
    model1_v2_cleaned_base_data.leave_time AS t1_leave_time,
    model1_v2_cleaned_base_data.route_number AS t1_route_number,
    model1_v2_cleaned_base_data.stop_time AS t1_stop_time,
    model1_v2_cleaned_base_data.arrive_time AS t1_arrive_time,
    model1_v2_cleaned_base_data.location_id AS t1_location_id,
    model1_v2_cleaned_base_data.schedule_status AS t1_schedule_status,
    model1_v2_cleaned_base_data.day_of_week AS t1_day_of_week,
    model1_v2_cleaned_base_data.dow_class AS t1_dow_class,
    model1_v2_cleaned_base_data.delay AS t1_delay
FROM
    model1_v2_base_model
    LEFT JOIN model1_v2_cleaned_base_data ON
        model1_v2_base_model.day_of_week = model1_v2_cleaned_base_data.
            day_of_week AND
        model1_v2_base_model.route_number = model1_v2_cleaned_base_data
            .route_number AND
        model1_v2_base_model.location_id = model1_v2_cleaned_base_data.
            location_id AND
        model1_v2_base_model.stop_time = model1_v2_cleaned_base_data.
            stop_time AND
        ABS(model1_v2_cleaned_base_data.delay) <= ABS(
            model1_v2_base_model.avg_delay) + model1_v2_base_model.
            std_delay;

-- VIEW NAME: model1_v2_final_res_agg
SELECT
    t2_day_of_week, t2_route_number,
    t2_location_id, t2_stop_time,
    t2_avg_delay, AVG(t1_delay) AS delay
FROM model1_v2_final_res_join
GROUP BY
    t2_day_of_week, t2_route_number,
```

```
    t2_location_id, t2_stop_time, t2_avg_delay;

-- VIEW NAME: model1_v2
SELECT
    t2_day_of_week AS day_of_week, t2_route_number AS route_number,
    t2_location_id AS location_id, t2_stop_time AS stop_time,
    CAST(COALESCE(delay, t2_avg_delay) AS INT) AS avg_delay
FROM model1_v2_final_res_agg;


-- STMT: 9
-- VIEW NAME: model1_v2_compare_sel_route
SELECT *
FROM model1_v2
WHERE route_number = 78;

-- VIEW NAME: model1_v2_compare_sel_dow_tue
SELECT *
FROM model1_v2_compare_sel_route
WHERE day_of_week = 2;

-- VIEW NAME: model1_v2_compare_sel_dow_wed
SELECT *
FROM model1_v2_compare_sel_route
WHERE day_of_week = 3;

-- VIEW NAME: model1_v2_compare_join
SELECT
    model1_v2_compare_sel_dow_tue.day_of_week AS t1_day_of_week,
    model1_v2_compare_sel_dow_tue.route_number AS t1_route_number,
    model1_v2_compare_sel_dow_tue.location_id AS t1_location_id,
    model1_v2_compare_sel_dow_tue.stop_time AS t1_stop_time,
```

```
    model1_v2_compare_sel_dow_tue.avg_delay AS t1_avg_delay,
    model1_v2_compare_sel_dow_wed.day_of_week AS t2_day_of_week,
    model1_v2_compare_sel_dow_wed.route_number AS t2_route_number,
    model1_v2_compare_sel_dow_wed.location_id AS t2_location_id,
    model1_v2_compare_sel_dow_wed.stop_time AS t2_stop_time,
    model1_v2_compare_sel_dow_wed.avg_delay AS t2_avg_delay
FROM
    model1_v2_compare_sel_dow_tue
    JOIN model1_v2_compare_sel_dow_wed ON
        model1_v2_compare_sel_dow_tue.location_id =
            model1_v2_compare_sel_dow_wed.location_id AND
        model1_v2_compare_sel_dow_tue.stop_time =
            model1_v2_compare_sel_dow_wed.stop_time;

-- VIEW NAME: model1_v2_compare_project
SELECT
    t1_route_number AS route_number, t1_location_id AS location_id,
    t1_stop_time AS stop_time, CAST(t1_avg_delay / 60 AS INT) AS
        dow1_delay,
    CAST(t2_avg_delay / 60 AS INT) AS dow2_delay
FROM model1_v2_compare_join;

-- VIEW NAME: model1_v2_compare
SELECT *
FROM model1_v2_compare_project
ORDER BY location_id, stop_time;


-- STMT: 10
-- VIEW NAME: model2_v2_cleaned_base_data
SELECT t1.*
FROM
```

```
264     model1_v2_select_base_data_with_delay AS t1,
265     (
266         SELECT
267             service_date, dow_class, route_number,
268             location_id, stop_time, min(delay) AS min_delay
269         FROM model1_v2_select_base_data_with_delay
270         GROUP BY
271             service_date, dow_class,
272             route_number, location_id, stop_time
273     ) AS t2
274 WHERE
275     t1.delay = t2.min_delay AND
276     t1.service_date = t2.service_date AND
277     t1.dow_class = t2.dow_class AND
278     t1.route_number = t2.route_number AND
279     t1.location_id = t2.location_id AND
280     t1.stop_time = t2.stop_time;
281
282 -- VIEW NAME: model2_v2_base_model
283 SELECT
284     dow_class, route_number, location_id,
285     stop_time, STDDEV(delay) AS std_delay,
286     AVG(delay) AS avg_delay
287 FROM model2_v2_cleaned_base_data
288 GROUP BY dow_class, route_number, location_id, stop_time;
289
290
291 -- VIEW NAME: model2_v2_final_res_join
292 SELECT
293     model2_v2_base_model.dow_class AS t2_dow_class,
294     model2_v2_base_model.route_number AS t2_route_number,
295     model2_v2_base_model.location_id AS t2_location_id,
```

```
296     model2_v2_base_model.stop_time AS t2_stop_time,
297     model2_v2_base_model.std_delay AS t2_std_delay,
298     model2_v2_base_model.avg_delay AS t2_avg_delay,
299     model2_v2_cleaned_base_data.service_date AS t1_service_date,
300     model2_v2_cleaned_base_data.leave_time AS t1_leave_time,
301     model2_v2_cleaned_base_data.route_number AS t1_route_number,
302     model2_v2_cleaned_base_data.stop_time AS t1_stop_time,
303     model2_v2_cleaned_base_data.arrive_time AS t1_arrive_time,
304     model2_v2_cleaned_base_data.location_id AS t1_location_id,
305     model2_v2_cleaned_base_data.schedule_status AS t1_schedule_status,
306     model2_v2_cleaned_base_data.day_of_week AS t1_day_of_week,
307     model2_v2_cleaned_base_data.dow_class AS t1_dow_class,
308     model2_v2_cleaned_base_data.delay AS t1_delay
309 FROM
310     model2_v2_base_model
311     LEFT JOIN model2_v2_cleaned_base_data ON
312         model2_v2_base_model.dow_class = model2_v2_cleaned_base_data.
                dow_class AND
313         model2_v2_base_model.route_number = model2_v2_cleaned_base_data
                .route_number AND
314         model2_v2_base_model.location_id = model2_v2_cleaned_base_data.
                location_id AND
315         model2_v2_base_model.stop_time = model2_v2_cleaned_base_data.
                stop_time AND
316         ABS(model2_v2_cleaned_base_data.delay) <= ABS(
                model2_v2_base_model.avg_delay) + model2_v2_base_model.
                std_delay;
317
318 -- VIEW NAME: model2_v2_final_res_agg
319 SELECT
320     t2_dow_class, t2_route_number,
321     t2_location_id, t2_stop_time,
```

```
    t2_avg_delay, AVG(t1_delay) AS delay
FROM model2_v2_final_res_join
GROUP BY
    t2_dow_class, t2_route_number,
    t2_location_id, t2_stop_time, t2_avg_delay;

-- VIEW NAME: model2_v2
SELECT
    t2_dow_class AS dow_class, t2_route_number AS route_number,
    t2_location_id AS location_id, t2_stop_time AS stop_time,
    CAST(COALESCE(delay, t2_avg_delay) AS INT) AS avg_delay
FROM model2_v2_final_res_agg;


-- STMT: 11
-- VIEW NAME: model2_v2_2_avg_delay_per_dow_class
SELECT *
FROM stop_events_with_dow
WHERE service_date >= '2018-11-01' AND service_date < '2019-02-01';

-- VIEW NAME: model2_v2_2_agg
SELECT
    dow_class, route_number, location_id, stop_time,
    AVG(arrive_time - stop_time) AS avg_delay_raw,
    count(*) AS num_of_observations
FROM model2_v2_2_avg_delay_per_dow_class
GROUP BY dow_class, route_number, location_id, stop_time;

-- VIEW NAME: model2_v2_2_proj
SELECT t1.*, CAST(avg_delay_raw AS INT) AS avg_delay
FROM model2_v2_2_agg AS t1;

```

```
354
355 -- STMT: 12
356 -- VIEW NAME: compare_v2_m1_m2_sel_m1
357 SELECT *
358 FROM model1_v2
359 WHERE
360     day_of_week = 5 AND
361     route_number in (76, 78) AND
362     location_id = 2285;
363
364 -- VIEW NAME: compare_v2_m1_m2_sel_m2
365 SELECT *
366 FROM model2_v2
367 WHERE dow_class = 'D';
368
369 -- VIEW NAME: compare_v2_m1_m2_join
370 SELECT
371     compare_v2_m1_m2_sel_m1.day_of_week AS t1_day_of_week,
372     compare_v2_m1_m2_sel_m1.route_number AS t1_route_number,
373     compare_v2_m1_m2_sel_m1.location_id AS t1_location_id,
374     compare_v2_m1_m2_sel_m1.stop_time AS t1_stop_time,
375     compare_v2_m1_m2_sel_m1.avg_delay AS t1_avg_delay,
376     compare_v2_m1_m2_sel_m2.dow_class AS t2_dow_class,
377     compare_v2_m1_m2_sel_m2.route_number AS t2_route_number,
378     compare_v2_m1_m2_sel_m2.location_id AS t2_location_id,
379     compare_v2_m1_m2_sel_m2.stop_time AS t2_stop_time,
380     compare_v2_m1_m2_sel_m2.avg_delay AS t2_avg_delay
381 FROM
382     compare_v2_m1_m2_sel_m1
383     JOIN compare_v2_m1_m2_sel_m2 ON
384         compare_v2_m1_m2_sel_m1.route_number = compare_v2_m1_m2_sel_m2.
              route_number AND
```

```
385         compare_v2_m1_m2_sel_m1.location_id = compare_v2_m1_m2_sel_m2.
                location_id AND
386         compare_v2_m1_m2_sel_m1.stop_time = compare_v2_m1_m2_sel_m2.
                stop_time;
387
388 -- VIEW NAME: compare_v2_m1_m2_project
389 SELECT
390     t1_route_number AS route_number,
391     t1_location_id AS location_id,
392     t1_stop_time AS stop_time,
393     CAST(t1_avg_delay / 60 AS INT) AS dow1_delay,
394     CAST(t2_avg_delay / 60 AS INT) AS dow2_delay
395 FROM compare_v2_m1_m2_join;
396
397 -- VIEW NAME: compare_v2_m1_m2
398 SELECT *
399 FROM compare_v2_m1_m2_project
400 ORDER BY location_id, stop_time;
401
402
403 -- STMT: 13
404 -- VIEW NAME: baseline_l1
405 SELECT *
406 FROM stop_events_with_dow
407 WHERE
408     service_date >= '2019-02-01' AND
409     service_date < '2019-03-01';
410
411 -- VIEW NAME: baseline_l3
412 SELECT delay, COUNT(*) AS observations
413 FROM baseline_l1
414 GROUP BY delay;
```

```
415
416 -- VIEW NAME: baseline_l4
417 SELECT
418     t1.*,
419     CASE
420         WHEN ABS(delay) > 5 THEN 'others'
421         ELSE CAST(delay AS STRING)
422     END AS delay_diffs
423 FROM baseline_l3 AS t1;
424
425 -- VIEW NAME: baseline_l6
426 SELECT delay_diffs, SUM(observations) AS observations
427 FROM baseline_l4
428 GROUP BY delay_diffs;
429
430 -- VIEW NAME: baseline_l7
431 SELECT delay_diffs, observations
432 FROM baseline_l6;
433
434 -- VIEW NAME: baseline_l8
435 SELECT *
436 FROM baseline_l7
437 ORDER BY delay_diffs;
438
439
440 -- STMT: 14
441 -- VIEW NAME: baseline_rush_hour_l1
442 SELECT *
443 FROM baseline_l1
444 WHERE
445     stop_time BETWEEN 23160 AND 31140 OR
446     stop_time BETWEEN 57600 AND 66780;
```

```sql

-- VIEW NAME: baseline_rush_hour_l4
SELECT delay, COUNT(*) AS observations
FROM baseline_rush_hour_l1
GROUP BY delay;

-- VIEW NAME: baseline_rush_hour_l5
SELECT *
FROM baseline_rush_hour_l4
ORDER BY observations DESC;


-- STMT: 15
-- VIEW NAME: predicting_feb_arrival_l1
SELECT *
FROM baseline_l1
WHERE route_number != 0 AND schedule_status != 6;

-- VIEW NAME: predicting_feb_arrival_l2
SELECT
    t1.*,
    CAST(CASE
        WHEN ABS(arrive_time - stop_time) <= ABS(leave_time -
            stop_time)
            THEN arrive_time - stop_time
        ELSE leave_time - stop_time
    END / 60 AS INT) AS actual_delay_in_min
FROM predicting_feb_arrival_l1 AS t1;

-- VIEW NAME: predicting_feb_arrival_l3
SELECT
    predicting_feb_arrival_l2.service_date AS t1_service_date,
```

```
478     predicting_feb_arrival_l2.leave_time AS t1_leave_time,
479     predicting_feb_arrival_l2.route_number AS t1_route_number,
480     predicting_feb_arrival_l2.stop_time AS t1_stop_time,
481     predicting_feb_arrival_l2.arrive_time AS t1_arrive_time,
482     predicting_feb_arrival_l2.location_id AS t1_location_id,
483     predicting_feb_arrival_l2.schedule_status AS t1_schedule_status,
484     predicting_feb_arrival_l2.day_of_week AS t1_day_of_week,
485     predicting_feb_arrival_l2.delay AS t1_delay,
486     predicting_feb_arrival_l2.dow_class AS t1_dow_class,
487     predicting_feb_arrival_l2.actual_delay_in_min AS
            t1_actual_delay_in_min,
488     model1_v2.day_of_week AS t2_day_of_week,
489     model1_v2.route_number AS t2_route_number,
490     model1_v2.location_id AS t2_location_id,
491     model1_v2.stop_time AS t2_stop_time,
492     model1_v2.avg_delay AS t2_avg_delay
493 FROM predicting_feb_arrival_l2
494     JOIN model1_v2 ON
495         predicting_feb_arrival_l2.route_number = model1_v2.route_number
                AND
496         predicting_feb_arrival_l2.location_id = model1_v2.location_id
               AND
497         predicting_feb_arrival_l2.stop_time = model1_v2.stop_time AND
498         predicting_feb_arrival_l2.day_of_week = model1_v2.day_of_week;
499
500 -- VIEW NAME: predicting_feb_arrival_l4
501 SELECT
502     t1.*,
503     CAST(t2_avg_delay / 60 AS INT) - t1_actual_delay_in_min AS
            delay_diff
504 FROM predicting_feb_arrival_l3 AS t1;
505
```

```sql
-- VIEW NAME: predicting_feb_arrival_l6
SELECT delay_diff, COUNT(*) AS observations
FROM predicting_feb_arrival_l4
GROUP BY delay_diff;

-- VIEW NAME: predicting_feb_arrival_l7
SELECT
    t1.*,
    CASE
        WHEN ABS(delay_diff) > 3 THEN 'others'
        ELSE CAST(delay_diff AS STRING)
    END AS delay_diffs
FROM predicting_feb_arrival_l6 AS t1;

-- VIEW NAME: predicting_feb_arrival_l10
SELECT delay_diffs, SUM(observations) AS observations
FROM predicting_feb_arrival_l7
GROUP BY delay_diffs;

-- VIEW NAME: predicting_feb_arrival_l11
SELECT *
FROM predicting_feb_arrival_l10
ORDER BY delay_diffs;


-- STMT: 16
-- VIEW NAME: predicting_feb_arrival_rush_hr_l1
SELECT *
FROM baseline_l1
WHERE
    stop_time BETWEEN 23160 AND 31140 OR
    stop_time BETWEEN 57600 AND 66780;
```

```
538
539 -- VIEW NAME: predicting_feb_arrival_rush_hr_l2
540 SELECT t1.*, CAST(delay / 60 AS INT) AS actual_delay_in_min
541 FROM predicting_feb_arrival_rush_hr_l1 AS t1;
542
543 -- VIEW NAME: predicting_feb_arrival_rush_hr_l3
544 SELECT
545     predicting_feb_arrival_rush_hr_l2.service_date AS t1_service_date,
546     predicting_feb_arrival_rush_hr_l2.leave_time AS t1_leave_time,
547     predicting_feb_arrival_rush_hr_l2.route_number AS t1_route_number,
548     predicting_feb_arrival_rush_hr_l2.stop_time AS t1_stop_time,
549     predicting_feb_arrival_rush_hr_l2.arrive_time AS t1_arrive_time,
550     predicting_feb_arrival_rush_hr_l2.location_id AS t1_location_id,
551     predicting_feb_arrival_rush_hr_l2.schedule_status AS
            t1_schedule_status,
552     predicting_feb_arrival_rush_hr_l2.day_of_week AS t1_day_of_week,
553     predicting_feb_arrival_rush_hr_l2.delay AS t1_delay,
554     predicting_feb_arrival_rush_hr_l2.dow_class AS t1_dow_class,
555     predicting_feb_arrival_rush_hr_l2.actual_delay_in_min AS
            t1_actual_delay_in_min,
556     model1_v2.day_of_week AS t2_day_of_week,
557     model1_v2.route_number AS t2_route_number,
558     model1_v2.location_id AS t2_location_id,
559     model1_v2.stop_time AS t2_stop_time,
560     model1_v2.avg_delay AS t2_avg_delay
561 FROM predicting_feb_arrival_rush_hr_l2
562     JOIN model1_v2 ON
563         predicting_feb_arrival_rush_hr_l2.route_number = model1_v2.
                route_number AND
564         predicting_feb_arrival_rush_hr_l2.location_id = model1_v2.
                location_id AND
```

```
565         predicting_feb_arrival_rush_hr_l2.stop_time = model1_v2.
                stop_time AND
566         predicting_feb_arrival_rush_hr_l2.day_of_week = model1_v2.
                day_of_week;
567
568 -- VIEW NAME: predicting_feb_arrival_rush_hr_l4
569 SELECT
570     t1.*,
571     CAST(t2_avg_delay / 60 AS INT) - t1_actual_delay_in_min AS
            delay_diff
572 FROM predicting_feb_arrival_rush_hr_l3 AS t1;
573
574 -- VIEW NAME: predicting_feb_arrival_rush_hr_l7
575 SELECT delay_diff, COUNT(*) AS observations
576 FROM predicting_feb_arrival_rush_hr_l4
577 GROUP BY delay_diff;
578
579 -- VIEW NAME: predicting_feb_arrival_rush_hr_l8
580 SELECT *
581 FROM predicting_feb_arrival_rush_hr_l7
582 ORDER BY observations DESC;
583
584
585 -- STMT: 17
586 -- VIEW NAME: predicting_feb_arrival_dow_class_l1
587 SELECT
588     predicting_feb_arrival_l2.service_date AS t1_service_date,
589     predicting_feb_arrival_l2.leave_time AS t1_leave_time,
590     predicting_feb_arrival_l2.route_number AS t1_route_number,
591     predicting_feb_arrival_l2.stop_time AS t1_stop_time,
592     predicting_feb_arrival_l2.arrive_time AS t1_arrive_time,
593     predicting_feb_arrival_l2.location_id AS t1_location_id,
```

```
    predicting_feb_arrival_l2.schedule_status AS t1_schedule_status,
    predicting_feb_arrival_l2.day_of_week AS t1_day_of_week,
    predicting_feb_arrival_l2.delay AS t1_delay,
    predicting_feb_arrival_l2.dow_class AS t1_dow_class,
    predicting_feb_arrival_l2.actual_delay_in_min AS
        t1_actual_delay_in_min,
    model2_v2_2_proj.dow_class AS t2_dow_class,
    model2_v2_2_proj.route_number AS t2_route_number,
    model2_v2_2_proj.location_id AS t2_location_id,
    model2_v2_2_proj.stop_time AS t2_stop_time,
    model2_v2_2_proj.avg_delay_raw AS t2_avg_delay_raw,
    model2_v2_2_proj.num_of_observations AS t2_num_of_observations,
    model2_v2_2_proj.avg_delay AS t2_avg_delay
FROM predicting_feb_arrival_l2
    JOIN model2_v2_2_proj ON
        predicting_feb_arrival_l2.route_number = model2_v2_2_proj.
            route_number AND
        predicting_feb_arrival_l2.location_id = model2_v2_2_proj.
            location_id AND
        predicting_feb_arrival_l2.stop_time = model2_v2_2_proj.
            stop_time AND
        predicting_feb_arrival_l2.dow_class = model2_v2_2_proj.
            dow_class;

-- VIEW NAME: predicting_feb_arrival_dow_class_l2
SELECT
    t1.*,
    CAST(t2_avg_delay / 60 AS INT) - t1_actual_delay_in_min AS
        delay_diff
FROM predicting_feb_arrival_dow_class_l1 AS t1;

-- VIEW NAME: predicting_feb_arrival_dow_class_l4
```

```sql
SELECT delay_diff, COUNT(*) AS observations
FROM predicting_feb_arrival_dow_class_l2
GROUP BY delay_diff;

-- VIEW NAME: predicting_feb_arrival_dow_class_l5
SELECT
    t1.*,
    CASE
        WHEN ABS(delay_diff) > 3 THEN 'others'
        ELSE CAST(delay_diff AS STRING)
    END AS delay_diffs
FROM predicting_feb_arrival_dow_class_l4 AS t1;

-- VIEW NAME: predicting_feb_arrival_dow_class_l8
SELECT delay_diffs, SUM(observations) AS observations
FROM predicting_feb_arrival_dow_class_l5
GROUP BY delay_diffs;

-- VIEW NAME: predicting_feb_arrival_dow_class_l9
SELECT *
FROM predicting_feb_arrival_dow_class_l8
ORDER BY delay_diffs;


-- STMT: 18
-- VIEW NAME: predicting_feb_arrival_rush_hr_dow_class_l1
SELECT
    predicting_feb_arrival_rush_hr_l2.service_date AS t1_service_date,
    predicting_feb_arrival_rush_hr_l2.leave_time AS t1_leave_time,
    predicting_feb_arrival_rush_hr_l2.route_number AS t1_route_number,
    predicting_feb_arrival_rush_hr_l2.stop_time AS t1_stop_time,
    predicting_feb_arrival_rush_hr_l2.arrive_time AS t1_arrive_time,
```

```
652     predicting_feb_arrival_rush_hr_l2.location_id AS t1_location_id,
653     predicting_feb_arrival_rush_hr_l2.schedule_status AS
            t1_schedule_status,
654     predicting_feb_arrival_rush_hr_l2.day_of_week AS t1_day_of_week,
655     predicting_feb_arrival_rush_hr_l2.delay AS t1_delay,
656     predicting_feb_arrival_rush_hr_l2.dow_class AS t1_dow_class,
657     predicting_feb_arrival_rush_hr_l2.actual_delay_in_min AS
            t1_actual_delay_in_min,
658     model2_v2_2_proj.dow_class AS t2_dow_class,
659     model2_v2_2_proj.route_number AS t2_route_number,
660     model2_v2_2_proj.location_id AS t2_location_id,
661     model2_v2_2_proj.stop_time AS t2_stop_time,
662     model2_v2_2_proj.avg_delay_raw AS t2_avg_delay_raw,
663     model2_v2_2_proj.num_of_observations AS t2_num_of_observations,
664     model2_v2_2_proj.avg_delay AS t2_avg_delay
665 FROM predicting_feb_arrival_rush_hr_l2
666     JOIN model2_v2_2_proj ON
667         predicting_feb_arrival_rush_hr_l2.route_number =
                model2_v2_2_proj.route_number AND
668         predicting_feb_arrival_rush_hr_l2.location_id =
                model2_v2_2_proj.location_id AND
669         predicting_feb_arrival_rush_hr_l2.stop_time = model2_v2_2_proj.
                stop_time AND
670         predicting_feb_arrival_rush_hr_l2.dow_class = model2_v2_2_proj.
                dow_class;
671
672 -- VIEW NAME: predicting_feb_arrival_rush_hr_dow_class_l2
673 SELECT
674     t1.*,
675     CAST(t2_avg_delay / 60 AS INT) - t1_actual_delay_in_min AS
            delay_diff
676 FROM predicting_feb_arrival_rush_hr_dow_class_l1 AS t1;
```

```sql

-- VIEW NAME: predicting_feb_arrival_rush_hr_dow_class_l5
SELECT delay_diff, COUNT(*) AS observations
FROM predicting_feb_arrival_rush_hr_dow_class_l2
GROUP BY delay_diff;

-- VIEW NAME: predicting_feb_arrival_rush_hr_dow_class_l6
SELECT *
FROM predicting_feb_arrival_rush_hr_dow_class_l5
ORDER BY observations DESC;


-- STMT: 19
-- VIEW NAME: model1_v3_l2
SELECT
    service_date, route_number, location_id, stop_time,
    MAX(arrive_time) AS arrive_time, MAX(leave_time) AS leave_time
FROM model1_v2_select_base_data
GROUP BY service_date, route_number, location_id, stop_time;

-- VIEW NAME: model1_v3_l3
SELECT t1.*, DAYOFWEEK(service_date) - 1 AS day_of_week
FROM model1_v3_l2 AS t1;

-- VIEW NAME: model1_v3_l5
SELECT
    day_of_week, route_number, location_id, stop_time,
    STDDEV(arrive_time) AS std_arrive_time,
    AVG(arrive_time) AS avg_arrive_time,
    STDDEV(leave_time) AS std_leave_time,
    AVG(leave_time) AS avg_leave_time
FROM model1_v3_l3
```

```sql
GROUP BY day_of_week, route_number, location_id, stop_time;

-- VIEW NAME: model1_v3_l6
SELECT
    model1_v3_l3.service_date AS t1_service_date,
    model1_v3_l3.route_number AS t1_route_number,
    model1_v3_l3.location_id AS t1_location_id,
    model1_v3_l3.stop_time AS t1_stop_time,
    model1_v3_l3.arrive_time AS t1_arrive_time,
    model1_v3_l3.leave_time AS t1_leave_time,
    model1_v3_l3.day_of_week AS t1_day_of_week,
    model1_v3_l5.day_of_week AS t2_day_of_week,
    model1_v3_l5.route_number AS t2_route_number,
    model1_v3_l5.location_id AS t2_location_id,
    model1_v3_l5.stop_time AS t2_stop_time,
    model1_v3_l5.std_arrive_time AS t2_std_arrive_time,
    model1_v3_l5.avg_arrive_time AS t2_avg_arrive_time,
    model1_v3_l5.std_leave_time AS t2_std_leave_time,
    model1_v3_l5.avg_leave_time AS t2_avg_leave_time
FROM model1_v3_l3
    JOIN model1_v3_l5 ON
        model1_v3_l5.day_of_week = model1_v3_l3.day_of_week AND
        model1_v3_l5.route_number = model1_v3_l3.route_number AND
        model1_v3_l5.location_id = model1_v3_l3.location_id AND
        model1_v3_l5.stop_time = model1_v3_l3.stop_time;

-- VIEW NAME: model1_v3_l8
SELECT
    t1.t2_day_of_week, t1.t2_route_number, t1.t2_location_id,
    t1.t2_stop_time, t1.t2_avg_arrive_time, t1.t2_avg_leave_time,
    t2.avg_arrive_time,
    t2.avg_leave_time
```

```
FROM
    (
        SELECT DISTINCT
            t2_day_of_week, t2_route_number, t2_location_id,
            t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time
        FROM model1_v3_l6
    ) AS t1
    LEFT JOIN (
        SELECT
            IFNULL(t3.t2_day_of_week, t4.t2_day_of_week) AS
                t2_day_of_week,
            IFNULL(t3.t2_route_number, t4.t2_route_number) AS
                t2_route_number,
            IFNULL(t3.t2_location_id, t4.t2_location_id) AS
                t2_location_id,
            IFNULL(t3.t2_stop_time, t4.t2_stop_time) AS t2_stop_time,
            IFNULL(t3.t2_avg_arrive_time, t4.t2_avg_arrive_time) AS
                t2_avg_arrive_time,
            IFNULL(t3.t2_avg_leave_time, t4.t2_avg_leave_time) AS
                t2_avg_leave_time,
            t3.avg_arrive_time,
            t4.avg_leave_time
        FROM (
            SELECT
                t2_day_of_week, t2_route_number, t2_location_id,
                t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time,
                AVG(t1_arrive_time) AS avg_arrive_time
            FROM model1_v3_l6
            WHERE
                abs(t1_arrive_time) <= abs(t2_avg_arrive_time) +
                    t2_std_arrive_time
            GROUP BY
```

```
767                 t2_day_of_week, t2_route_number, t2_location_id,
768                 t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time
769         ) AS t3
770         FULL JOIN (
771             SELECT
772                 t2_day_of_week, t2_route_number, t2_location_id,
773                 t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time,
774                 AVG(t1_leave_time) AS avg_leave_time
775             FROM model1_v3_l6
776             WHERE
777                 abs(t1_leave_time) <= abs(t2_avg_leave_time) +
                        t2_std_leave_time
778             GROUP BY
779                 t2_day_of_week, t2_route_number, t2_location_id,
780                 t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time
781         ) AS t4 ON
782             t3.t2_day_of_week = t4.t2_day_of_week AND
783             t3.t2_route_number = t4.t2_route_number AND
784             t3.t2_location_id = t4.t2_location_id AND
785             t3.t2_stop_time = t4.t2_stop_time AND
786             t3.t2_avg_arrive_time = t4.t2_avg_arrive_time AND
787             t3.t2_avg_leave_time = t4.t2_avg_leave_time
788     ) AS t2 ON
789         t1.t2_day_of_week = t2.t2_day_of_week AND
790         t1.t2_route_number = t2.t2_route_number AND
791         t1.t2_location_id = t2.t2_location_id AND
792         t1.t2_stop_time = t2.t2_stop_time AND
793         t1.t2_avg_arrive_time = t2.t2_avg_arrive_time AND
794         t1.t2_avg_leave_time = t2.t2_avg_leave_time;
795
796
797 -- VIEW NAME: model1_v3_l9
```

```sql
SELECT
    t2_day_of_week AS day_of_week, t2_route_number AS route_number,
    t2_location_id AS location_id, t2_stop_time AS stop_time,
    CAST(COALESCE(avg_arrive_time, t2_avg_arrive_time) AS INT) AS
        arrive_time,
    CAST(COALESCE(avg_leave_time,t2_avg_leave_time) AS INT) AS
        leave_time
FROM model1_v3_l8;



-- STMT: 20
-- VIEW NAME: model2_v3_l1
SELECT t1.*,
    CASE
        WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
        WHEN day_of_week = 0 THEN 'U'
        ELSE 'S'
    END AS dow_class
FROM model1_v3_l3 AS t1;

-- VIEW NAME: model2_v3_l3
SELECT
    dow_class, route_number, location_id, stop_time,
    STDDEV(arrive_time) AS std_arrive_time,
    AVG(arrive_time) AS avg_arrive_time,
    STDDEV(leave_time) AS std_leave_time,
    AVG(leave_time) AS avg_leave_time
FROM model2_v3_l1
GROUP BY dow_class, route_number, location_id, stop_time;

```

```sql
-- VIEW NAME: model2_v3_l4
SELECT
    model2_v3_l1.service_date AS t1_service_date,
    model2_v3_l1.route_number AS t1_route_number,
    model2_v3_l1.location_id AS t1_location_id,
    model2_v3_l1.stop_time AS t1_stop_time,
    model2_v3_l1.arrive_time AS t1_arrive_time,
    model2_v3_l1.leave_time AS t1_leave_time,
    model2_v3_l1.day_of_week AS t1_day_of_week,
    model2_v3_l1.dow_class AS t1_dow_class,
    model2_v3_l3.dow_class AS t2_dow_class,
    model2_v3_l3.route_number AS t2_route_number,
    model2_v3_l3.location_id AS t2_location_id,
    model2_v3_l3.stop_time AS t2_stop_time,
    model2_v3_l3.std_arrive_time AS t2_std_arrive_time,
    model2_v3_l3.avg_arrive_time AS t2_avg_arrive_time,
    model2_v3_l3.std_leave_time AS t2_std_leave_time,
    model2_v3_l3.avg_leave_time AS t2_avg_leave_time
FROM model2_v3_l1
    JOIN model2_v3_l3 ON
        model2_v3_l1.dow_class = model2_v3_l3.dow_class AND
        model2_v3_l1.route_number = model2_v3_l3.route_number AND
        model2_v3_l1.location_id = model2_v3_l3.location_id AND
        model2_v3_l1.stop_time = model2_v3_l3.stop_time;


-- VIEW NAME: model2_v3_l6
SELECT
    t1.t2_dow_class, t1.t2_route_number, t1.t2_location_id,
    t1.t2_stop_time, t1.t2_avg_arrive_time, t1.t2_avg_leave_time,
    t2.avg_arrive_time,
    t2.avg_leave_time
```

```
FROM
    (
        SELECT DISTINCT
            t2_dow_class, t2_route_number, t2_location_id,
            t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time
        FROM model2_v3_l4
    ) AS t1
    LEFT JOIN (
        SELECT
            IFNULL(t3.t2_dow_class, t4.t2_dow_class) AS t2_dow_class,
            IFNULL(t3.t2_route_number, t4.t2_route_number) AS
                t2_route_number,
            IFNULL(t3.t2_location_id, t4.t2_location_id) AS
                t2_location_id,
            IFNULL(t3.t2_stop_time, t4.t2_stop_time) AS t2_stop_time,
            IFNULL(t3.t2_avg_arrive_time, t4.t2_avg_arrive_time) AS
                t2_avg_arrive_time,
            IFNULL(t3.t2_avg_leave_time, t4.t2_avg_leave_time) AS
                t2_avg_leave_time,
            t3.avg_arrive_time,
            t4.avg_leave_time
        FROM
            (
                SELECT
                    t2_dow_class, t2_route_number, t2_location_id,
                    t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time
                        ,
                    AVG(t1_arrive_time) AS avg_arrive_time
                FROM model2_v3_l4
                WHERE
                    abs(t1_arrive_time) <= abs(t2_avg_arrive_time) +
                        t2_std_arrive_time
```

```
                GROUP BY
                    t2_dow_class, t2_route_number, t2_location_id,
                    t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time
            ) AS t3
            FULL JOIN (
                SELECT
                    t2_dow_class, t2_route_number, t2_location_id,
                    t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time
                        ,
                    AVG(t1_leave_time) AS avg_leave_time
                FROM model2_v3_l4
                WHERE
                    abs(t1_leave_time) <= abs(t2_avg_leave_time) +
                        t2_std_leave_time
                GROUP BY
                    t2_dow_class, t2_route_number, t2_location_id,
                    t2_stop_time, t2_avg_arrive_time, t2_avg_leave_time
            ) AS t4 ON
                t3.t2_dow_class = t4.t2_dow_class AND
                t3.t2_route_number = t4.t2_route_number AND
                t3.t2_location_id = t4.t2_location_id AND
                t3.t2_stop_time = t4.t2_stop_time AND
                t3.t2_avg_arrive_time = t4.t2_avg_arrive_time AND
                t3.t2_avg_leave_time = t4.t2_avg_leave_time
    ) AS t2 ON
        t1.t2_dow_class = t2.t2_dow_class AND
        t1.t2_route_number = t2.t2_route_number AND
        t1.t2_location_id = t2.t2_location_id AND
        t1.t2_stop_time = t2.t2_stop_time AND
        t1.t2_avg_arrive_time = t2.t2_avg_arrive_time AND
        t1.t2_avg_leave_time = t2.t2_avg_leave_time;

```

```
916
917 -- VIEW NAME: model2_v3_l7
918 SELECT
919     t2_dow_class AS dow_class, t2_route_number AS route_number,
920     t2_location_id AS location_id, t2_stop_time AS stop_time,
921     CAST(COALESCE(avg_arrive_time, t2_avg_arrive_time) AS INT) AS
          arrive_time,
922     CAST(COALESCE(avg_leave_time,t2_avg_leave_time) AS INT) AS
          leave_time
923 FROM model2_v3_l6;
924
925
926 -- STMT: 21
927 -- VIEW NAME: baseline_v2_l1
928 SELECT *
929 FROM baseline_l1
930 WHERE route_number <> 0;
931
932 -- VIEW NAME: baseline_v2_l3
933 SELECT
934     service_date, route_number, location_id, stop_time,
935     MIN(arrive_time) AS arrive_time, MAX(leave_time) AS leave_time
936 FROM baseline_v2_l1
937 GROUP BY service_date, route_number, location_id, stop_time;
938
939 -- VIEW NAME: baseline_v2_l4
940 SELECT
941     CAST(CASE
942         WHEN ABS(arrive_time - stop_time) <= ABS(leave_time - 30 -
              stop_time)
943             THEN arrive_time - stop_time
944         ELSE leave_time - 30 - stop_time
```

```
    END / 60 AS INT) AS prediction_diff
FROM baseline_v2_l3;

-- VIEW NAME: baseline_v2_l5
SELECT
    CASE
        WHEN prediction_diff > 3 THEN 'others'
        ELSE CAST(prediction_diff AS STRING)
    END AS prediction_diffs
FROM baseline_v2_l4;

-- VIEW NAME: baseline_v2_l8
SELECT prediction_diffs, COUNT(*) AS observations
FROM baseline_v2_l5
GROUP BY prediction_diffs;

-- VIEW NAME: baseline_v2_l9
SELECT *
FROM baseline_v2_l8
ORDER BY prediction_diffs;



-- STMT: 22
-- VIEW NAME: baseline_v2_rush_hour_l1
SELECT *
FROM baseline_rush_hour_l1
WHERE route_number <> 0;

-- VIEW NAME: baseline_v2_rush_hour_l3
SELECT
    service_date, route_number, location_id, stop_time,
```

```
977     MIN(arrive_time) AS arrive_time, MAX(leave_time) AS leave_time
978 FROM baseline_v2_rush_hour_l1
979 GROUP BY service_date, route_number, location_id, stop_time;
980
981 -- VIEW NAME: baseline_v2_rush_hour_l4
982 SELECT
983     CAST(CASE
984         WHEN ABS(arrive_time - stop_time) <= ABS(leave_time - 30 -
                stop_time)
985             THEN arrive_time - stop_time
986         ELSE leave_time - 30 - stop_time
987     END / 60 AS INT) AS prediction_diff
988 FROM baseline_v2_rush_hour_l3;
989
990 -- VIEW NAME: baseline_v2_rush_hour_l5
991 SELECT
992     CASE
993         WHEN prediction_diff > 3 THEN 'others'
994         ELSE CAST(prediction_diff AS STRING)
995     END AS prediction_diffs
996 FROM baseline_v2_rush_hour_l4;
997
998 -- VIEW NAME: baseline_v2_rush_hour_l8
999 SELECT prediction_diffs, COUNT(*) AS observations
1000 FROM baseline_v2_rush_hour_l5
1001 GROUP BY prediction_diffs;
1002
1003 -- VIEW NAME: baseline_v2_rush_hour_l9
1004 SELECT *
1005 FROM baseline_v2_rush_hour_l8
1006 ORDER BY prediction_diffs;
1007
```

```sql


-- STMT: 23
-- VIEW NAME: comp_predic_v2_l1
SELECT t1.*, DAYOFWEEK(service_date) - 1 AS day_of_week
FROM baseline_v2_l3 AS t1;

-- VIEW NAME: comp_predic_v2_l2
SELECT
    comp_predic_v2_l1.service_date AS t1_service_date,
    comp_predic_v2_l1.route_number AS t1_route_number,
    comp_predic_v2_l1.location_id AS t1_location_id,
    comp_predic_v2_l1.stop_time AS t1_stop_time,
    comp_predic_v2_l1.arrive_time AS t1_arrive_time,
    comp_predic_v2_l1.leave_time AS t1_leave_time,
    comp_predic_v2_l1.day_of_week AS t1_day_of_week,
    model1_v3_l9.day_of_week AS t2_day_of_week,
    model1_v3_l9.route_number AS t2_route_number,
    model1_v3_l9.location_id AS t2_location_id,
    model1_v3_l9.stop_time AS t2_stop_time,
    model1_v3_l9.arrive_time AS t2_arrive_time,
    model1_v3_l9.leave_time AS t2_leave_time
FROM comp_predic_v2_l1
    JOIN model1_v3_l9 ON
        comp_predic_v2_l1.route_number = model1_v3_l9.route_number AND
        comp_predic_v2_l1.location_id = model1_v3_l9.location_id AND
        comp_predic_v2_l1.stop_time = model1_v3_l9.stop_time AND
        comp_predic_v2_l1.day_of_week = model1_v3_l9.day_of_week;

-- VIEW NAME: comp_predic_v2_l3
SELECT
    t1.*,
```

```
1040     CAST((t2_arrive_time - t1_arrive_time) / 60 AS INT) AS
             prediction_diff
1041 FROM comp_predic_v2_l2 AS t1;
1042
1043 -- VIEW NAME: comp_predic_v2_l5
1044 SELECT prediction_diff, COUNT(*) AS observations
1045 FROM comp_predic_v2_l3
1046 GROUP BY prediction_diff;
1047
1048 -- VIEW NAME: comp_predic_v2_l6
1049 SELECT t1.*, CASE
1050     WHEN prediction_diff > 3 THEN 'others'
1051     ELSE CAST(prediction_diff AS STRING)
1052     END AS prediction_diffs
1053 FROM comp_predic_v2_l5 AS t1;
1054
1055 -- VIEW NAME: comp_predic_v2_l9
1056 SELECT prediction_diffs, SUM(observations) AS observations
1057 FROM comp_predic_v2_l6
1058 GROUP BY prediction_diffs;
1059
1060 -- VIEW NAME: comp_predic_v2_l10
1061 SELECT *
1062 FROM comp_predic_v2_l9
1063 ORDER BY prediction_diffs;
1064
1065
1066
1067
1068 -- STMT: 24
1069 -- VIEW NAME: comp_predic_v2_rush_hour_l1
1070 SELECT t1.*, DAYOFWEEK(service_date) - 1 AS day_of_week
```

```sql
FROM baseline_v2_rush_hour_l3 AS t1;

-- VIEW NAME: comp_predic_v2_rush_hour_l2
SELECT
    comp_predic_v2_rush_hour_l1.service_date AS t1_service_date,
    comp_predic_v2_rush_hour_l1.route_number AS t1_route_number,
    comp_predic_v2_rush_hour_l1.location_id AS t1_location_id,
    comp_predic_v2_rush_hour_l1.stop_time AS t1_stop_time,
    comp_predic_v2_rush_hour_l1.arrive_time AS t1_arrive_time,
    comp_predic_v2_rush_hour_l1.leave_time AS t1_leave_time,
    comp_predic_v2_rush_hour_l1.day_of_week AS t1_day_of_week,
    model1_v3_l9.day_of_week AS t2_day_of_week,
    model1_v3_l9.route_number AS t2_route_number,
    model1_v3_l9.location_id AS t2_location_id,
    model1_v3_l9.stop_time AS t2_stop_time,
    model1_v3_l9.arrive_time AS t2_arrive_time,
    model1_v3_l9.leave_time AS t2_leave_time
FROM comp_predic_v2_rush_hour_l1
    JOIN model1_v3_l9 ON
        comp_predic_v2_rush_hour_l1.route_number = model1_v3_l9.
            route_number AND
        comp_predic_v2_rush_hour_l1.location_id = model1_v3_l9.
            location_id AND
        comp_predic_v2_rush_hour_l1.stop_time = model1_v3_l9.stop_time
            AND
        comp_predic_v2_rush_hour_l1.day_of_week = model1_v3_l9.
            day_of_week;

-- VIEW NAME: comp_predic_v2_rush_hour_l3
SELECT
    t1.*,
```

```
    CAST((t2_arrive_time - t1_arrive_time) / 60 AS INT) AS
        prediction_diff
FROM comp_predic_v2_rush_hour_l2 AS t1;

-- VIEW NAME: comp_predic_v2_rush_hour_l5
SELECT prediction_diff, COUNT(*) AS observations
FROM comp_predic_v2_rush_hour_l3
GROUP BY prediction_diff;

-- VIEW NAME: comp_predic_v2_rush_hour_l6
SELECT t1.*, CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE CAST(prediction_diff AS STRING)
    END AS prediction_diffs
FROM comp_predic_v2_rush_hour_l5 AS t1;

-- VIEW NAME: comp_predic_v2_rush_hour_l9
SELECT prediction_diffs, SUM(observations) AS observations
FROM comp_predic_v2_rush_hour_l6
GROUP BY prediction_diffs;

-- VIEW NAME: comp_predic_v2_rush_hour_l10
SELECT *
FROM comp_predic_v2_rush_hour_l9
ORDER BY prediction_diffs;


-- STMT: 25
-- VIEW NAME: comp_predic_v3_l1
SELECT t1.*, CASE
    WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
    WHEN day_of_week = 0 THEN 'U'
```

```
    ELSE 'S'
    END AS dow_class
FROM comp_predic_v2_l1 AS t1;

-- VIEW NAME: comp_predic_v3_l2
SELECT
    comp_predic_v3_l1.service_date AS t1_service_date,
    comp_predic_v3_l1.route_number AS t1_route_number,
    comp_predic_v3_l1.location_id AS t1_location_id,
    comp_predic_v3_l1.stop_time AS t1_stop_time,
    comp_predic_v3_l1.arrive_time AS t1_arrive_time,
    comp_predic_v3_l1.leave_time AS t1_leave_time,
    comp_predic_v3_l1.day_of_week AS t1_day_of_week,
    comp_predic_v3_l1.dow_class AS t1_dow_class,
    model2_v3_l7.dow_class AS t2_dow_class,
    model2_v3_l7.route_number AS t2_route_number,
    model2_v3_l7.location_id AS t2_location_id,
    model2_v3_l7.stop_time AS t2_stop_time,
    model2_v3_l7.arrive_time AS t2_arrive_time,
    model2_v3_l7.leave_time AS t2_leave_time
FROM comp_predic_v3_l1
    JOIN model2_v3_l7 ON
        comp_predic_v3_l1.route_number = model2_v3_l7.route_number AND
        comp_predic_v3_l1.location_id = model2_v3_l7.location_id AND
        comp_predic_v3_l1.stop_time = model2_v3_l7.stop_time AND
        comp_predic_v3_l1.dow_class = model2_v3_l7.dow_class;

-- VIEW NAME: comp_predic_v3_l3
SELECT
    t1.*,
    CAST((t2_arrive_time - t1_arrive_time) / 60 AS INT) AS
        prediction_diff
```

```
FROM comp_predic_v3_l2 AS t1;

-- VIEW NAME: comp_predic_v3_l5
SELECT prediction_diff, COUNT(*) AS observations
FROM comp_predic_v3_l3
GROUP BY prediction_diff;

-- VIEW NAME: comp_predic_v3_l6
SELECT t1.*, CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE CAST(prediction_diff AS STRING)
    END AS prediction_diffs
FROM comp_predic_v3_l5 AS t1;

-- VIEW NAME: comp_predic_v3_l9
SELECT prediction_diffs, SUM(observations) AS observations
FROM comp_predic_v3_l6
GROUP BY prediction_diffs;

-- VIEW NAME: comp_predic_v3_l10
SELECT *
FROM comp_predic_v3_l9
ORDER BY prediction_diffs;



-- STMT: 26
-- VIEW NAME: comp_predic_v3_rush_hour_l1
SELECT t1.*, CASE
    WHEN day_of_week IN (1,2,3,4,5) THEN 'D'
    WHEN day_of_week = 0 THEN 'U'
    ELSE 'S'
```

```sql
    END AS dow_class
FROM comp_predic_v2_rush_hour_l1 AS t1;

-- VIEW NAME: comp_predic_v3_rush_hour_l2
SELECT
    comp_predic_v3_rush_hour_l1.service_date AS t1_service_date,
    comp_predic_v3_rush_hour_l1.route_number AS t1_route_number,
    comp_predic_v3_rush_hour_l1.location_id AS t1_location_id,
    comp_predic_v3_rush_hour_l1.stop_time AS t1_stop_time,
    comp_predic_v3_rush_hour_l1.arrive_time AS t1_arrive_time,
    comp_predic_v3_rush_hour_l1.leave_time AS t1_leave_time,
    comp_predic_v3_rush_hour_l1.day_of_week AS t1_day_of_week,
    comp_predic_v3_rush_hour_l1.dow_class AS t1_dow_class,
    model2_v3_l7.dow_class AS t2_dow_class,
    model2_v3_l7.route_number AS t2_route_number,
    model2_v3_l7.location_id AS t2_location_id,
    model2_v3_l7.stop_time AS t2_stop_time,
    model2_v3_l7.arrive_time AS t2_arrive_time,
    model2_v3_l7.leave_time AS t2_leave_time
FROM comp_predic_v3_rush_hour_l1
    JOIN model2_v3_l7 ON
        comp_predic_v3_rush_hour_l1.route_number = model2_v3_l7.
            route_number AND
        comp_predic_v3_rush_hour_l1.location_id = model2_v3_l7.
            location_id AND
        comp_predic_v3_rush_hour_l1.stop_time = model2_v3_l7.stop_time
            AND
        comp_predic_v3_rush_hour_l1.dow_class = model2_v3_l7.dow_class;

-- VIEW NAME: comp_predic_v3_rush_hour_l3
SELECT
    t1.*,
```

```
    CAST((t2_arrive_time - t1_arrive_time) / 60 AS INT) AS
        prediction_diff
FROM comp_predic_v3_rush_hour_l2 AS t1;

-- VIEW NAME: comp_predic_v3_rush_hour_l5
SELECT prediction_diff, COUNT(*) AS observations
FROM comp_predic_v3_rush_hour_l3
GROUP BY prediction_diff;

-- VIEW NAME: comp_predic_v3_rush_hour_l6
SELECT t1.*, CASE
    WHEN prediction_diff > 3 THEN 'others'
    ELSE CAST(prediction_diff AS STRING)
    END AS prediction_diffs
FROM comp_predic_v3_rush_hour_l5 AS t1;

-- VIEW NAME: comp_predic_v3_rush_hour_l9
SELECT prediction_diffs, SUM(observations) AS observations
FROM comp_predic_v3_rush_hour_l6
GROUP BY prediction_diffs;

-- VIEW NAME: comp_predic_v3_rush_hour_l10
SELECT *
FROM comp_predic_v3_rush_hour_l9
ORDER BY prediction_diffs;


-- STMT: 27
-- VIEW NAME: comp_pred_model1_and_model2_l1
SELECT
    model1_v3_l9.day_of_week AS t1_day_of_week,
    model1_v3_l9.route_number AS t1_route_number,
```

```
    model1_v3_l9.location_id AS t1_location_id,
    model1_v3_l9.stop_time AS t1_stop_time,
    model1_v3_l9.arrive_time AS t1_arrive_time,
    model1_v3_l9.leave_time AS t1_leave_time,
    model2_v3_l7.dow_class AS t2_dow_class,
    model2_v3_l7.route_number AS t2_route_number,
    model2_v3_l7.location_id AS t2_location_id,
    model2_v3_l7.stop_time AS t2_stop_time,
    model2_v3_l7.arrive_time AS t2_arrive_time,
    model2_v3_l7.leave_time AS t2_leave_time
FROM model1_v3_l9
    JOIN model2_v3_l7 ON
        model1_v3_l9.route_number = model2_v3_l7.route_number AND
        model1_v3_l9.location_id = model2_v3_l7.location_id AND
        model1_v3_l9.stop_time = model2_v3_l7.stop_time;

-- VIEW NAME: comp_pred_model1_and_model2_l2
SELECT *
FROM comp_pred_model1_and_model2_l1
WHERE
    t1_day_of_week = 5 AND
    t2_dow_class = 'D' AND
    t1_route_number in (76, 78) AND
    t1_location_id = 2285;

-- VIEW NAME: comp_pred_model1_and_model2_l3
SELECT
    t1_route_number AS route_number,
    t1_location_id AS location_id,
    t1_stop_time AS stop_time,
     t1_arrive_time AS model1_pred_arrival_time,
    t1_leave_time AS model1_pred_leave_time,
```

```
    t2_arrive_time AS model2_pred_arrival_time,
    t2_leave_time AS model2_pred_leave_time
FROM comp_pred_model1_and_model2_l2;

-- VIEW NAME: comp_pred_model1_and_model2_l4
SELECT *
FROM comp_pred_model1_and_model2_l3
ORDER BY location_id, stop_time;
```

A.7.1 Min-Max Queries

The min-max queries that we used in Spark are exactly the same as those we did for MySQL in Section A.5.1.